Broadview

安全技术大系



看雪软件安全

GOOD

畅销书升级版

加密与解密

第三版

段钢 编著

揭示软件加密与解密最核心

看雪安全技术团队全力支持



电子工业出版社

VISIT...

LANZAROTE Caliente.com

目　录

作者简介
前言
第1篇　基础篇
第1章　基础知识
1.1　文本字符
1.1.1　字节存储顺序
1.1.2　ASCII与Unicode字符集
1.2　Windows操作系统
1.2.1　Win API简介
1.2.2　常用Win32 API函数
1.2.3　什么是句柄
1.2.4　Windows 9x与Unicode
1.2.5　Windows NT/2000/XP与Unicode
1.2.6　Windows消息机制
1.3　保护模式简介
1.3.1　虚拟内存
1.3.2　保护模式的权限级别
1.4　认识PE格式
第2篇　调试篇
第2章　动态分析技术
2.1　OllyDbg 调试器
2.1.1　OllyDbg界面
2.1.2　OllyDbg的配置
2.1.3　加载程序
2.1.4　基本操作
2.1.5　断点
2.1.6　插件
2.1.7　Run trace
2.1.8　Hit trace
2.1.9　符号调试技术
2.1.10　OllyDbg常见问题
2.2　SoftICE调试器

第3章　静态分析技术
3.1　文件类型分析
3.1.1　PEiD工具
3.1.2　FileInfo工具
3.2　静态反汇编
3.2.1　反汇编引擎
3.2.2　IDA Pro简介
3.2.3　IDA的配置
3.2.4　IDA主窗口界面
3.2.5　交叉参考
3.2.6　参考重命名
3.2.7　标签的用法
3.2.8　进制的转换
3.2.9　代码和数据转换
3.2.10　字符串
3.2.11　数组
3.2.12　结构体
3.2.13　枚举类型
3.2.14　堆栈变量
3.2.15　IDC脚本
3.2.16　FLIRT
3.2.17　插件
3.2.18　其他功能
3.2.19　小结
3.3　可执行文件的修改
3.4　静态分析技术应用实例
3.4.1　解密初步
3.4.2　逆向工程初步
第4章　逆向分析技术
4.1　启动函数
4.2　函数
4.2.1　函数的识别
4.2.2　函数的参数
4.2.3　函数的返回值
4.3　数据结构
4.3.1　局部变量
4.3.2　全局变量

4.3.3 数组
4.4 虚函数
4.5 控制语句
4.5.1 IF-THEN-ELSE语句
4.5.2 SWITCH-CASE语句
4.5.3 转移指令机器码的计算
4.5.4 条件设置指令（SETcc）
4.5.5 纯算法实现逻辑判断
4.6 循环语句
4.7 数学运算符
4.7.1 整数的加法和减法
4.7.2 整数的乘法
4.7.3 整数的除法
4.8 文本字符串
4.8.1 字符串存储格式
4.8.2 字符寻址指令
4.8.3 字母大小写转换
4.8.4 计算字符串的长度
4.9 指令修改技巧
第3篇 解密篇
第5章 常见的演示版保护技术
5.1 序列号保护方式
5.1.1 序列号保护机制
5.1.2 如何攻击序列号保护
5.1.3 字符串比较形式
5.1.4 注册机制作
5.2 警告（Nag）窗口
5.3 时间限制
5.3.1 计时器
5.3.2 时间限制
5.3.3 拆解时间限制保护
5.4 菜单功能限制
5.4.1 相关函数
5.4.2 拆解菜单限制保护
5.5 KeyFile保护
5.5.1 相关API函数

5.5.2 拆解KeyFile保护
5.6 网络验证
5.6.1 相关函数
5.6.2 网络验证破解一般思路
5.7 CD-Check
5.7.1 相关函数
5.7.2 拆解光盘保护
5.8 只运行一个实例
5.8.1 实现方法
5.8.2 实例
5.9 常用断点设置技巧
第6章 加密算法
6.1 单向散列算法
6.1.1 MD5算法
6.1.2 SHA算法
6.1.3 小结
6.2 对称加密算法
6.2.1 RC4流密码
6.2.2 TEA算法
6.2.3 IDEA算法
6.2.4 BlowFish算法
6.2.5 AES算法
6.2.6 对称加密算法小结
6.3 公开密钥加密算法
6.3.1 RSA算法
6.3.2 ElGamal公钥算法
6.3.3 DSA数字签名算法
6.3.4 椭圆曲线密码编码学（Elliptic Curve Cryptography）
6.4 其他算法
6.4.1 CRC32算法
6.4.2 Base64编码
6.5 常见的加密库接口及其识别
6.5.1 Miracl大数运算库
6.5.2 FGInt
6.5.3 其他加密算法库介绍
第4篇 语言和平台篇

第7章　Delphi程序
7.1　DeDe反编译器
7.2　按钮事件代码
7.3　模块初始化与结束化
第8章　Visual Basic程序
8.1　基础知识
8.1.1　字符编码方式
8.1.2　编译模式
8.2　自然编译（Native）
8.2.1　相关VB函数
8.2.2　VB程序比较方式
8.3　伪编译
8.3.1　虚拟机与伪代码
8.3.2　动态分析VB P-code程序
8.3.3　伪代码的综合分析
8.3.4　VB P-code攻击实战
第9章　.Net平台加解密
9.1　.Net概述
9.1.1　什么是.Net
9.1.2　几个基本概念
9.1.3　第一个.Net程序
9.2　MSIL与元数据
9.2.1　PE结构的扩展
9.2.2　.Net下的汇编MSIL
9.2.3　MSIL与元数据的结合
9.3　代码分析技术
9.3.1　静态分析
9.3.2　动态调试
9.3.3　代码修改
9.4　代码保护技术及其逆向
9.4.1　强名称
9.4.2　名称混淆
9.4.3　流程混淆
9.4.4　压缩
9.4.5　加密
9.4.6　其他保护手段

9.5 深入.Net
9.5.1 反射与CodeDOM
9.5.2 Unmaganed API
9.5.3 Rotor、MONO与.Net内核
第5篇 系统篇
第10章 PE文件格式
10.1 PE的基本概念
10.1.1 基地址
10.1.2 相对虚拟地址
10.1.3 文件偏移地址
10.2 MS-DOS头部
10.3 PE文件头
10.3.1 Signature字段
10.3.2 IMAGE_FILE_HEADER结构
10.3.3 IMAGE_OPTIONAL_HEADER结构
10.4 区块
10.4.1 区块表
10.4.2 各种区块的描述
10.4.3 区块的对齐值
10.4.4 文件偏移与虚拟地址转换
10.5 输入表
10.5.1 输入函数的调用
10.5.2 输入表结构
10.5.3 输入地址表（IAT）
10.5.4 输入表实例分析
10.6 绑定输入
10.7 输出表
10.7.1 输出表结构
10.7.2 输出表结构实例分析
10.8 基址重定位
10.8.1 基址重定位概念
10.8.2 基址重定位结构定义
10.8.3 基址重定位结构实例分析
10.9 资源
10.9.1 资源结构
10.9.2 资源结构实例分析

10.9.3　资源编辑工具
10.10　TLS初始化
10.11　调试目录
10.12　延迟装入数据
10.13　程序异常数据
10.14　.Net头部
10.15　编写PE分析工具
10.15.1　文件格式检查
10.15.2　FileHeader和OptionalHeader内容的读取
10.15.3　得到数据目录表信息
10.15.4　得到区块表信息
10.15.5　得到输出表信息
10.15.6　得到输入表信息
第11章　结构化异常处理
11.1　基本概念
11.1.1　异常列表
11.1.2　异常处理的基本过程
11.1.3　SEH的分类
11.2　SEH相关数据结构
11.2.1　TEB结构
11.2.2　EXCEPTION_REGISTRATION结构
11.2.3　EXCEPTION_POINTERS、EXCEPTION RECORD、CONTEXT
11.3　异常处理回调函数
第6篇　脱壳篇
第12章　专用加密软件
12.1　认识壳
12.1.1　壳的概念
12.1.2　压缩引擎
12.2　压缩壳
12.2.1　UPX
12.2.2　ASPack
12.3　加密壳
12.3.1　ASProtect
12.3.2　Armadillo
12.3.3　EXECryptor

12.3.4 Themida
12.4 虚拟机保护软件
12.4.1 虚拟机介绍
12.4.2 VMProtect简介
第13章 脱壳技术
13.1 基础知识
13.1.1 壳的加载过程
13.1.2 脱壳机
13.1.3 手动脱壳
13.2 寻找OEP
13.2.1 根据跨段指令寻找OEP
13.2.2 用内存访问断点找OEP
13.2.3 根据堆栈平衡原理找OEP
13.2.4 根据编译语言特点找OEP
13.3 抓取内存映像
13.3.1 Dump原理
13.3.2 反Dump技术（Anti-Dump）
13.4 重建输入表
13.4.1 输入表重建的原理
13.4.2 确定IAT的地址和大小
13.4.3 根据IAT重建输入表
13.4.4 ImportREC重建输入表
13.4.5 输入表加密概括
13.5 DLL文件脱壳
13.5.1 寻找OEP
13.5.2 Dump映像文件
13.5.3 重建DLL的输入表
13.5.4 构造重定位表
13.6 附加数据
13.7 PE文件的优化
13.8 压缩壳
13.8.1 UPX外壳
13.8.2 ASPack外壳
13.9 加密壳
13.9.1 ASProtect
13.9.2 Themida的SDK分析

13.10　静态脱壳
13.10.1　外壳Loader的分析
13.10.2　编写静态脱壳器
第7篇　保护篇
第14章　软件保护技术
14.1　防范算法求逆
14.1.1　基本概念
14.1.2　堡垒战术
14.1.3　游击战术
14.2　抵御静态分析
14.2.1　花指令
14.2.2　SMC技术实现
14.2.3　信息隐藏
14.2.4　简单的多态变形技术
14.3　文件完整性检验
14.3.1　磁盘文件校验实现
14.3.2　校验和（Checksum）
14.3.3　内存映像校验
14.4　代码与数据结合技术
14.4.1　准备工作
14.4.2　加密算法选用
14.4.3　手动加密代码
14.4.4　使.text区块可写
14.5　软件保护的若干忠告
第15章　反跟踪技术
15.1　由BeingDebugged引发的蝴蝶效应
15.1.1　BeingDebugged
15.1.2　NtGlobalFlag
15.1.3　Heap Magic
15.1.4　从源头消灭BeingDebugged
15.2　回归Native：用户态的梦魇
15.2.1　CheckRemoteDebuggerPresent
15.2.2　ProcessDebugPort
15.2.3　ThreadHideFromDebugger
15.2.4　Debug Object
15.2.5　SystemKernelDebuggerInformation

15.2.6 Native API
15.2.7 Hook和AntiHook
15.3 真正的奥秘：小技巧一览
15.3.1 SoftICE检测方法
15.3.2 OllyDbg检测方法
15.3.3 调试器漏洞
15.3.4 防止调试器附加
15.3.5 父进程检测
15.3.6 时间差
15.3.7 通过Trap Flag检测
15.3.8 双进程保护
第16章 外壳编写基础
16.1 外壳的结构
16.2 加壳主程序
16.2.1 判断文件是否为PE格式
16.2.2 文件基本数据读入
16.2.3 附加数据读取
16.2.4 输入表处理
16.2.5 重定位表处理
16.2.6 文件的压缩
16.2.7 资源数据处理
16.2.8 区块的融合
16.3 外壳部分编写
16.3.1 外壳的加载过程
16.3.2 自建输入表
16.3.3 外壳引导段
16.3.4 外壳第二段
16.4 将外壳部分添加至原程序
第17章 虚拟机的设计
17.1 原理
17.1.1 反汇编引擎
17.1.2 指令分类
17.2 启动框架和调用约定
17.2.1 调度器VStartVM
17.2.2 虚拟环境：VMContext
17.2.3 平衡堆栈：VBegin和VCheckEsp
17.3 Handler的设计

17.3.1 辅助Handler
17.3.2 普通Handler和指令拆解
17.3.3 标志位问题
17.3.4 相同作用的指令
17.3.5 转移指令
17.3.6 转移跳转指令的另一种实现
17.3.7 call指令
17.3.8 retn指令
17.3.9 不可模拟指令
17.4 托管代码的异常处理
17.4.1 VC++的异常处理
17.4.2 Delphi的异常处理
17.5 小结
第8篇 PEDIY篇
第18章 补丁技术
18.1 文件补丁
18.2 内存补丁
18.2.1 跨进程内存存取机制
18.2.2 Debug API机制
18.2.3 利用调试寄存器机制
18.2.4 DLL劫持技术
18.3 SMC补丁
18.3.1 单层SMC补丁技术
18.3.2 多层SMC补丁技术
18.4 补丁工具
第19章 代码的二次开发
19.1 数据对齐
19.2 增加空间
19.2.1 区块间隙
19.2.2 手工构造区块
19.2.3 工具辅助构造区块
19.3 获得函数的调用
19.3.1 增加输入函数
19.3.2 显式链接调用DLL
19.4 代码的重定位
19.4.1 修复重定位表

19.4.2　代码的自定位技术
19.5　增加输出函数
19.6　消息循环
19.6.1　WndProc函数
19.6.2　寻找消息循环
19.6.3　WndProc汇编形式
19.7　修改WndProc扩充功能
19.7.1　扩充WndProc
19.7.2　扩充Exit菜单功能
19.7.3　扩充Open菜单功能
19.8　增加接口
19.8.1　用DLL增加功能
19.8.2　扩展消息循环
附录A　浮点指令
附录B　在Visual C++中使用内联汇编
术语表
参考文献

安全技术大系

[image]

看雪软件安全
http://www.pediy.com

加密与解密
第三版

段钢　编著

電子工業出版社
Publishing House of Electronics Industry
北京·**BEIJING**

内容简介

本书以加密与解密为切入点，讲述了软件安全领域许多基础知识和技能，如调试技能、逆向分析、加密保护、外壳开发、虚拟机设计等。读者在掌握本书的内容时，很容易在漏洞分析、安全编程、病毒分析、软件保护等领域扩展，这些知识点都是相互的，彼此联系。国内高校对软件安全这块领域教育重视程度还不够，许多方面还是空白，而近年来社会和企业对软件安全技术人才需求逐年上升。从就业角度来说，掌握这方面技术，可以提高自身的职场竞争能力；从个人成长角度来说，研究软件安全技术有助于掌握许多系统底层知识，是提升职业技能的重要途径。作为一名合格的程序员，除了掌握需求分析、设计模式等外，如能掌握一些系统底层知识，熟悉整个系统的底层结构，对自己的工作必将获益良多。

本书可以作为大中专学校或培训机构的软件安全辅助教材，是安全技术爱好者、调试人员、程序开发人员不可多得的一本好书。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（**CIP**）数据

加密与解密/段钢编著．—3版．—北京：电子工业出版社，2008.7
（安全技术大系）
ISBN 978-7-121-06644-3

I．加…　II．段…　III．电子计算机 - 密码术　IV．TP309.7

中国版本图书馆CIP数据核字（2008）第064470号

策划编辑：郭　立
责任编辑：葛　娜
印　　刷：北京中新伟业印刷有限公司
装　　订：
出版发行：电子工业出版社
北京市海淀区万寿路173信箱　邮编100036
开　　本：880×1230　1/16　印张：35.75　字数：1018千字
印　　次：2011年7月第8次印刷
印　　数：26001 ~ 28000册　　定价：59.00元

http://pan.baidu.com/share/link?shareid=512559&uk=1077227309&third=15　密码:ocj7

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

作者简介

本书由看雪软件安全网站（看雪学院）站长段钢主持编著。在本书的编写过程中，参与创作的每位作者倾力将各自擅长的专业技术毫无保留地奉献给广大读者，使得本书展现出了极具价值的丰富内容。如果读者在阅读本书后，能够感受到管窥技术奥秘带来的内心的喜悦，并愿意与大家分享这份感受，这是作者最大的愿望。

主编：段钢

编委：（按章节顺序排列）

Blowfish，沈晓斌，丁益青，单海波，王勇，赵勇，唐植明，softworm，afanty，李江涛，林子深，印豪，冯典，罗翼，林小华，郭春杨

编委档案

Blowfish

看雪首席版主。经验丰富的大龄程序员。1992年上大学始接触电脑，1997年读研期间接触网络并自学加密与解密技术，一发不可收拾，其时常在教育网BBS灌水。喜多方涉猎，亦能抓住一点深入钻研，对逆向分析技术尤为痴迷。多年来常在看雪论坛灌水，见证了论坛的风风雨雨，也结识了一些不错的朋友。

参与章节：第5章　5.1　序列号保护方式
第14章　14.5　软件保护的若干忠告

沈晓斌

看雪核心专家团队成员。看雪论坛ID为cnbragon，现攻读密码学专业硕士学位。最初的爱好是网络安全，进而研究软件的逆向工程，对密码学的兴趣由此而发。对密码学的各个方面都有所涉猎，尤其擅长密码学在软件保护中的应用研究。独立完成了一个加密算法库CryptoFBC。译作有《程序员密码学》。

个人主页：www.cnbragon.cn

参与章节：第6章　加密算法

丁益青

看雪技术专家。看雪论坛ID为cyclotron，复旦大学在读硕士研究生，复旦大学日月光华BBS黑客与系统安全版版主，致力于Windows环境下可执行文件的加密解密与逆向工程研究。主要作品有EmbedPE、IDT Protector、PEunLOCK等。

个人主页：cyclotron.ycublog.com

参与章节：第8章　8.3　伪编译

单海波

看雪核心专家团队成员。看雪论坛ID为tankaiha，生于六朝古都南京，硕士研究生毕业，现任某研究所工程师，工作之余好与计算机为伴。2002年接触汇编并热衷于病毒技术学习，后偶遇看雪学院，遂终日游戏于程序加密与解密，不可自拔。2006年与kanxue及坛中数位好友成立.net安全小组DST（Dotnet Reverse Team），共同探讨.net平台下的软件安全技术。

个人主页：http://vxer.en/blog

参与章节：第9章　.Net平台加解密

王　勇

看雪技术专家。毕业于石油大学（华东）计算机科学与技术专业。擅长C/C＋＋、ASM和驱动程序开发。对面向对象程序设计和Windows系统底层的研究有丰富的经验。很高兴这次能与各位高手一起合作，也希望能与编程爱好者及加密解密爱好者更多的交流。

主页：http://www.w-yong.com

参与章节：第10章　10.15　编写PE分析工具

赵　勇

看雪技术专家。来自江苏江阴，计算机业余爱好者，兴趣爱好广泛。

参与章节：第13章　13.6　附加数据

唐植明

看雪技术核心权威。看雪论坛ID为DiKeN，2002年毕业于兰州大学，计算机科学与技术专业。爱好逆向工程，iPB（inside Pandora's Box）组织创始人（在这儿更是要感谢组织的兄弟姐妹们，大家团结友好，互相学习，为iPB的成功做出了巨大努力），曾在2002年编写过《加密与解密实战攻略》算法部分。

参与章节：第13章　13.10　静态脱壳

softworm

看雪技术天才。70后一代，非计算机专业的业余爱好者。1998年开始接触逆向与破解，迄今已近10年，终于达到了“知道自己不知道”的境界。感兴趣的方向包括壳、虚拟机保护、病毒引擎、Rootkit。后两项还处于只知道名字的水平，愿与有共同爱好的朋友们一起学习。

E-mail:softworm2003@hotmail.com

参与章节：第13章　13.9.2　Themida的SDK分析

afanty

看雪技术专家。多年专业研究软件加/解密技术。

参与章节：第14章　14.1　防范算法求逆

李江涛

看雪技术核心权威。看雪论坛ID为ljtt，喜欢学习编程技术，常用编程语言为VC/MASM。对PB、VFP的反编译有深入的研究，写过DePB、

FoxSpy等程序。平时大多数时间都在电脑上耕作，最大的希望是能够领悟到编程的精髓，写一个自己比较满意的作品。

E-mail:shellfan@163.com

参与章节：第14章　14.2.2　SMC技术实现

林子深

看雪技术导师。看雪论坛ID为forgot，1989年生，看雪论坛外壳开发小组组长。熟悉Win32平台和80x86汇编，擅长代码的逆向，对壳的研究比较多。

E-mail:forgot@live.com

参与章节：第12章　12.4.1　虚拟机介绍
第14章　14.2.4　简单的多态变形技术
第15章　反跟踪技术

印　豪

看雪资深技术权威。看雪论坛ID为Hying，擅长加壳技术，拥有独立创作的加密利器。

E-mail:newhying001@163.com

参与章节：第16章　外壳编写基础

冯　典

看雪技术天才。看雪论坛ID为bughoho，1990年生，来自四川，看雪论坛虚拟机开发小组组长，目前工作主要是从事逆向研究。

个人自述：记得14岁时家里买了台电脑，使我对编程有了极大的兴趣。16岁上高一时已对读书彻底不感兴趣，于是退学（现在的我才发现，我并不是对读书不感兴趣，而是对教育制度的反感）。后来听了家人的意见，转读四川新华电脑学校，感受颇多，一月之后便退学，至于为什么我就不说了。17岁时，一个偶然的机会，使我对逆向有了浓厚的兴趣，并接触到看雪论坛，也认识到了kanxue。承蒙kanxue抬举，让我执笔虚拟机这一章，由于我并不是一个才高八斗的人，所以写得也没有那么的妙笔生花、鬼斧神工了。

参与章节：第17章　虚拟机的设计

罗　翼

看雪技术专家。资深程序员，由加/解密知识起接触编程，对Windows底层机制有多年的研究经验。后由于工作需要，接触C＋＋/ATL/COM等技术。现致力于研究各种Moder C＋＋的元素的应用范围及其对降低程序复杂度所起的作用，热切关注ISO C＋＋以及分布式计算相关内容的进展。

参与章节：第18章　18.2.1　跨进程内存存取机制
18.2.2　Debug API机制
18.2.3　利用调试寄存器机制

林小华

看雪资深版主。看雪论坛ID为linhanshi，武汉大学电力系统及其自

动化专业，『工具分区』区版主，对论坛的工具版块发展做出了重大贡献。

个人主页：http://blog.csdn.net/linhanshi

参与章节：第12章　12.3　加密壳

第18章　18.4　补丁工具

郭春杨

看雪技术专家。看雪论坛ID为Yonsm，软件工程师，从事视频编解码和多媒体软件设计工作。对Windows和Windows Mobile系统有比较深入的了解。

主页：WWW.Yonsm.NET

E-mail:Yonsm@163.com

参与章节：附录B　在Visual C++中使用内联汇编

前　言

软件安全是信息安全领域的重要内容，涉及到软件相关的加密、解密、逆向分析、漏洞分析、安全编程以及病毒分析等。目前，国内高校对软件安全教育重视程度不够，许多方面还是空白。随着互联网应用的普及和企业信息化程度的不断提升，社会和企业对软件安全技术人才需求逐年上升，在计算机病毒查杀、网游安全、网络安全、个人信息安全等方面人才缺口很大，相关职位待遇较高。从就业角度来看，掌握软件安全相关知识和技能，不但可以提高自身的职场竞争能力，而且有机会发挥更大的个人潜力，获得满意的薪酬；从个人成长方面来说，研究软件安全技术有助于掌握许多系统底层知识，是提升职业技能的重要途径。作为一名合格的程序员，除了掌握需求分析、设计模式等外，如能掌握一些系统底层知识，熟悉整个系统的底层结构，对自己的工作必将获益良多。

本书以软件加密与解密为切入点，讲述了软件安全领域相关基础知识和技能。读者在阅读了本书的内容后，很容易在漏洞分析、安全编程、病毒分析等领域得到扩展。这些知识点的相互关联性，将促使读者开阔思路，使所学融会贯通，领悟更多的学习方法，提升自身学习能力。

本书是《加密与解密》的第三版，此书今天能够与读者见面，完全是广大读者的热情和鼓舞带来的成果，作者深表谢意。

关于看雪学院

本书作者是软件安全主题网站——“看雪学院”的站长。看雪软件安全网站（www.pediy.com）由kanxue（作者网名）创建于2000年。网站历经8年多的发展，脱颖而出，凭借自身实力，已经成为中国软件安全领域公认的最权威的技术站点，影响深远。

2000年初，笔者想找一些研究软件加解密的朋友交流一下，但十分令人遗憾的是，那时国内这方面的技术资料很缺乏，不成系统，大家的交流也十分有限。因此，笔者自己建立了一个主页“看雪学院”，期望与兴趣相投的朋友共同探讨加密与解密的知识。当初这个简单的网站，就是今天看雪软件安全网站的雏形，并且是当时国内唯一从技术角度研究软件加/解密的站点。很短的时间，这个站点就获得了大家的认同，并在广大网友的支持下，健康地成长起来。随着我们的努力，网站推出的软件调试论坛逐渐成为国内知名度最高的软件安全论坛，吸引了众多高手。

本着知识共享，一切免费的建站宗旨，看雪软件安全网站汇聚了大量高水平的技术文章，至今为止原创了数千余篇精华文章，极大地推动了国内软件安全技术的发展。2007年论坛改名为看雪软件安全论坛，论坛在保持已有的软件加密与解密研究方面外，在漏洞分析、系统底层、病毒分析、Rootkit等技术领域进行全面扩展，逐步发展为信息安全的综

合服务网站。

多年来，看雪软件安全网站一直遵循纯技术的发展策略，不但在行业中树立了令人尊敬的专业形象，更使一大批专业人士和专家聚集在这里，形成了一个技术交流的网上家园，带动了大批对软件安全感兴趣的网友加入进来，构建起了一个围绕软件安全主题的活跃的大社区，历久弥新。正是看到这种技术气氛，不少知名的公司都很关注论坛技术人才，如微软公司信息安全部门、珠海金山毒霸公司、深圳腾讯公司、360安全中心、启明星辰以及部分网游公司等。

为了推进软件安全技术为社会和企业服务的理念，我们正在努力提升看雪网站的社会作用和价值，从而为关注信息安全的大众，提供更好的服务和技术产品。

看雪软件安全网站，汇聚了许多志趣相投的朋友，经历了风风雨雨的8年，一直走到今天实属不易。作为网站站长和此书的作者，本人在此由衷地感谢所有关心和支持我们的共同事业，参与共同发展的朋友们！每到网站最困难的时候，是你们伸出无私的援助之手，才让网站渡过了一个个难关，能有今天的大好局面！在此特别鸣谢以下朋友和机构的大力支持：

海城金航网络科技有限公司阿男为网站提供网站空间

雅联网络服务有限责任公司李智勇为论坛提供独立服务器

南京慧速科技发展有限公司刘小荣为论坛提供独立服务器

感谢陈超达为服务器安全维护所做的大量工作

本书的缘起

当今的信息社会里，安全技术越来越重要了，如何普及软件安全知识是作者始终关注的一个大问题。正是为了更好地将软件安全知识普及到社会各个领域的愿望，促成了本书的问世。

依托看雪学院的技术背景，由作者主编和主导的看雪软件安全系列书籍，目前已出版发行了《加密与解密——软件保护技术及完全解决方案》（简体版，繁体版）、《加密与解密（第二版）》（简体版，繁体版）、《软件加密技术内幕》等书籍；基于电子资料的形式，历年发行的《看雪论坛精华》被众多网站转载，保守计算，其下载量已经超过数百万份，极大地推动了国内软件安全技术的发展。

[image]

[image]

这是一本很难写的书，因为2000年时，软件安全是一个全新的领域。从Windows 95面世以来的6年内，市面上没有一本这方面的书，网上也缺乏相关资料。为了填补国内Windows平台上加密与解密书籍的空白，作者与看雪论坛的一流好手努力合作，克服种种困难，于2001年9月推出了国内第一本全面介绍Windows平台下软件加密与解密技术的书籍，这就是本书的第一版《加密与解密——软件保护技术及完全解决方案》。

在第一版中，我们试图从软件加密和解密这两个方面对当今流行的软件保护技术进行分析。希望读者看过此书之后，能够对各种流行的软件保护与破解技术有所了解。

第一版一面世就得到了广大读者的喜爱和认可，获得了2002年全国优秀畅销书奖（科技类）！在全国很多计算机专业书店获得了名列前茅的销售业绩，而且一年来在著名的华储网销售排行中都被排在前几名内。次年，本书在台湾发行了繁体版，得到了台湾读者的热烈欢迎。

[image]

2003年6月以本书第一版为基础，完成了本书的第二版《加密与解密》。

笔者从2004年开始第三版的更新准备工作，这个版本编写时间比较长，前后用了四年多的时间才得已完稿。这是所有参与者共同的努力，是他们把自己才华中最精彩部分展现给大家了。

现在读者看到的这本500多页的图书，几乎包含了当今Windows 32位环境下软件保护技术的绝大部分内容，从基本的跟踪调试到深层的拆解脱壳；从浅显的分析注册到中高级软件保护与分析，其跨度之广、内容之深，国内至今尚无同类出版物能与之比肩。

第三版的变化

第三版是在《加密与解密》第二版与《软件加密技术内幕》两本书的基础上完成的，删除了第二版中的过时内容，将《软件加密技术内幕》一些知识点补充融合进来，结构更加合理。

1．讲解通俗，突出基础

本书加强了基础部分的篇幅，系统讲解软件逆向的整个基本流程，包括动态分析、静态分析，以及逆向分析的基础知识。比如重点讲解了逆向必备工具OllyDbg和IDA的用法，并详细讲述逆向分析的基础知识，初学者通过相关几章的学习，可以轻松入门。

2．案例丰富，覆盖面广

书中提供了大量的案例分析，方便读者理论与实践相结合。通过实际操作，提高读者的调试分析能力。

3．加强了密码学算法

密码学算法越来越多地应用在软件保护领域，调试软件必须对比较知名的密码学算法有一定的了解。“加密算法”这一章，讲解了常见密码学算法的应用。

4．新增**.Net**技术

随着微软.Net平台的推广，越来越多的开发者开始关注.Net程序的安全。.Net这章向读者普及了.Net安全的基本知识。

5．加强脱壳基础知识的篇幅

脱壳一章的结构和内容规划，参考了大量的建议，组织更加合理，完全为脱壳新手量身定做。

6．软件保护技术实施

相关章节详细研究了大量极具商业价值的保护技术，包括反跟踪技术、外壳编写基础、虚拟机的设计等，读者完全可以将这些技术应用到自己软件保护之中去。

7．二次开发与补丁技术

“代码的二次开发”一章中讲解如何在没有源码的情况下，扩充程序功能，打造开发接口；“补丁技术”一章讲解如何自己编程实现内存补丁或内存注册机。

本书预备知识

在阅读本书前，读者应该对汇编语言有大致的了解。汇编语言是大学计算机的必修课，这方面的书籍品种很多，如《IBM PC汇编语言程序设计》，虽然大多数书以DOS汇编为讲解平台，但对理解汇编指令功能依然有益。

读者如果熟悉和了解C语言，对阅读本书是很有帮助的。

建议掌握一些Win32编程，不论研究加密与解密，还是编程，都应该了解Win32编程。Win32编程是API方式的Windows程序设计，学习Windows API能使读者更深入地了解Windows工作方式。此类书籍推荐您阅读Charles Petzold所著的《Windows程序设计》，该书堪称经典之作，它以C语言为讲解平台。

到此为止，作者将不再假设你已经具有任何加/解密的经验了。

适合的读者

本书适合以下读者：

• 软件安全技术相关工作者：本书是软件安全研究的一本不错的技术字典；

• 对调试技术感兴趣的读者：提高读者的调试技能，增强软件的质量；

• 对软件保护感兴趣的软件开发人员：更好地保护你的作品；

• 大中专在校学生：通过本书掌握的相关知识和技能，将使你获得职场竞争的秘密武器；

• 其他：关注个人信息安全、计算机安全技术，并且想了解技术内幕的朋友，可以从中获得答案。

内容导读

大多数人可能认为软件加密与解密是一门高深的学问。造成这种认识的原因是以前这方面的技术资料缺乏，从而将“加密与解密”这一技术“神”化了。初学者一般不知从何下手，由于没方向，花费了大量时间

和精力，走了不少弯路。本书给对这方面感兴趣的读者指明一个方向，提供一个捷径。

本书大部分章节既关联又彼此独立，因此读者可以根据自己情况，选择合适自己的内容阅读。

[image]

[image]

[image]

特别致谢

首先真诚感谢我的父母、妻子、女儿对我的大力支持，使得我顺利完成此书的编写！我所有的荣耀都属于你们。

谨此对电子工业出版社博文视点公司所有相关人员致以真诚的谢意！

特别感谢电子工业出版社博文视点公司总经理郭立所做的大量工作！

特别感谢上海盛大网络发展有限公司徐海峡、王峰、刘庆民、蒋渭华、李明、张子雁、史昕峰、彭伟、张静盛等对本书的大力支持！

特别感谢微软公司大中华区首席安全官江明灶和微软的战略安全架构专家裔云天对本书的支持！

特别感谢珠海金山毒霸事业部陈勇、赵闽的技术支持！

特别感谢看雪软件安全论坛核心管理团队CCDebuger、Ivanov、riijj、michael的支持！

特别感谢看雪软件安全论坛各版主及各技术小组成员，对本书的大力支持！他们是：

（1）北极星2003、笨笨雄、crackabcer、cnbragon、linhanshi、LOVE、monkeycz、逍遥风、小虾、zmworm

（2）软件调试小组：aker、hawking、elance、theOcrat

（3）虚拟机技术小组：bughoho、linxer、wangdell、Isaiah

（4）外壳开发小组：forgot、dummy、bithaha

（5）工具开发小组：doskey、netsowell、freecat、wak、menting

（6）编程技术小组：北极星2003、没有风、CCDeath、Combojiang、Sislcb

（7）PTG翻译小组：arhat、thinkSJ、kkbing、aalloverred、月中人、alpsdew、jdxyw、Jhlqb、mjahuolong

（8）.Net小组：tankaiha、backer、dreaman、inraining、kkbing、lccracker、oep1、rick、slan、tracky、菩提！、MegaX

感谢CCDebuger对“第2章　动态分析技术”和“第13章　脱壳技术”校对！

感谢gzgzlxg对“第3章　静态分析技术”提出的修正和补充意见！

感谢zmworm对工具IDA使用的补充建议！

感谢Intel公司中国企业应用技术支持部的段夕华对“第4章　逆向分

析技术”提出的宝贵修正意见！

感谢WiNrOOt翻译的www.datarescue.com提供的IDA简易教程，IDA部分参考了一下！

感谢riijj为“5.6　网络验证”一节提供的实例！

感谢cnbragon参与的“第6章　加密算法”！

感谢cyclotron参与的“8.3　伪编译”！

感谢tankaiha参与的“第9章　.Net平台加解密”！

感谢Hume对“第11章　结构化异常处理”提供的技术支持！

感谢DiKeN参与的“13.10　静态脱壳”！

感谢softworm参与的“13.9.2　Themida的SDK分析”！

感谢forgot参与的“14.2.4　简单的多态变形技术”、“第15章　反跟踪技术”！

感谢Hying参与的“第16章　外壳编写基础”！

感谢bughoho参与的“第17章　虚拟机的设计”！

感谢afanty参与的“14.1　防范算法求逆”！

感谢并参考老罗（www.luocong.com）“矛与盾的较量——CRC实践篇”！

感谢Lenus在内存Dump和内存断点方面给予的技术支持！

感谢TiANWEi翻译的SoftICE手册！

感谢wynney签名制作的帮助！

感谢skylly为脱壳一章提供的脚本制作的技术支持！

感谢hnhuqiong提供的ODbgScript脚本教学！

感谢linhanshi在工具方面提供的帮助！

感谢VolX为本书配套光盘映像文件提供的Aspr2.XX_unpacker.osc脚本！

感谢CoDe_Inject对“18.2.4　DLL劫持技术”一节提供的帮助！

感谢武汉科锐软件培训中心（www.51asm.com）Backer为“18.2.4　DLL劫持技术”提供lpk.cpp！

感谢frozenrain、jero、mocha、NWMonster、petnt、sudami、tankaiha、wynney、XPoy、王清、小虾等朋友为术语表所做的工作！

感谢Sun Bird、JoJo、kvllz等人对本书的大力支持！

感谢fonge等诸多看雪论坛会员持续一年多来的发帖签名支持新书！

同时，也要感谢那些共同参与《加密与解密》（第一、二版）、《软件加密技术内幕》组稿的看雪软件安全论坛的众多一流好手，是他们的参与和奉献才让此书得以顺利完成。

这次的第三版改动较大，参考引用了如下朋友在《加密与解密》（第一、二版）中的文章：

（1）Blowfish在第一版参与的“第5章　软件保护技术”；

（2）Fisheep参与的“浮点指令小结”和“信息隐藏技术”；

（3）吴朝相（http://www.souxin.com）参与的“认识壳”；

（4）mr.wei参与的“DeDe用法”；

（5）感谢pll621在扩展PE功能开拓性的研究；

（6）娃娃（王凌迪）提供的“MD5算法”资料。

参考并引用了如下朋友在《软件加密技术内幕》中的文章：

（1）Hying的Anti_Dump；
（2）Hume的“第4章　Windows下的异常处理”；
（3）王勇的“编写PE分析工具”；
（4）罗翼的“3.3　利用调试API制作内存补丁”；
（5）郭春杨的“在Visual C＋＋中使用内联汇编”；
（6）Ljtt参与的“花指令”、“SMC技术实现”、“壳的加载过程”；
（7）dREAMtHEATER翻译的Matt Pietrek An In-Depth Look into the Win32 Portable Executable File Format。

在此，还要感谢看雪软件安全论坛其他朋友的支持和帮助！是你们提供的帮助，才使得我能够完成此书。如果以上未提及对您的谢意，在此，我表示由衷的感谢！

关于本书配套光盘映像文件

本书不提供配套光盘，光盘映像文件可以到本站主页下载。

由于版权问题，光盘映像文件仅提供书中提到的免费软件或共享软件。如果从学习角度需要使用那些有版权的软件，建议读者通过搜索引擎查找。

光盘映像文件提供的软件经过多方面检查测试，绝无病毒。但一些加/解密工具采用了某些病毒技术，因此部分代码与某些病毒的特征码类似，会造成查毒软件的误报。请自行决定使用。

建议将光盘映像中的文件拷贝到硬盘，并去除只读属性再调试，以免出现一些无法解释的错误。

光盘映像文件下载：http://book.pediy.com

反馈信息

我们非常希望能够了解读者对本书的看法。如果您有什么问题或自己的学习心得，欢迎发到看雪软件安全网站——看雪学院。

技术支持：http://www.pediy.com

邮件地址：kanxue@pediy.com

段　钢

2008.5.1于上海

第1篇　基础篇

■ **第1章　基础知识**

什么是API？什么是Unicode？Windows 9x与Windows 2000/XP上的加密/解密究竟有什么不同？只有了解这些基础知识，在加密与解密过程中才能有的放矢地处理各种问题。本书的第1章将系统地解答这些问题。

第1章　基础知识

研究加密与解密，必须要了解一些Windows系统的基础知识，这样在分析的过程中才能有的放矢地处理各种问题。

1.1　文本字符

在学习过程中会与各类字符打交道，它们在Windows里扮演着重要角色。

1.1.1　字节存储顺序

多字节数据是按怎样的顺序存放的呢？实际情况和CPU有关，微处理机中的存放顺序有正序（Big-Endian）和逆序（Little-Endian）之分。常见的Intel体系芯片使用的编码方式属于Little-Endian类；某些RISC架构的CPU，如IBM的power-PC等属于Big-Endian类。

两种编码区别：

- Big-Endian　　高位字节存入低地址，低位字节存入高地址，依次排列；
- Little-Endian　低位字节存入低地址，高位字节存入高地址，反序排列。

例如，将12345678h写入到以1000h开始的内存中，则结果如图1.1所示。本书以运行在Intel x86 CPU上的Windows为讲解平台，因此涉及的编码皆为Little-Endian类。

[image]

图1.1　Big-Endian与Little-Endian内存存储方式

1.1.2　ASCII与Unicode字符集

美国信息交换标准码（ASCII）是一个7位的编码标准，包括26个小写字母、26个大写字母、10个数字、32个符号、33个控制代码和一个空格，总共128个代码。由于计算机通常用“字节”（byte）这个8位的存储单位来进行信息交换，因此不同的计算机厂家对ASCII进行了扩充，增加了128个附加的字符来补充ASCII，它们的值在127以上的部分是不统一的。例如ANSI，Symbol，OEM等字符集，其中ANSI是系统预设的标准文字存储格式。表1-1列出了用十六进制数（Hex）与十进制数（Dec）表示的部分常用字符的ASCII值。

表**1-1**　常用字符的**ASCII**值

[image]

Unicode是ASCII字符编码的一个扩展。只不过在Windows中，用两个字节对其进行编码，也称为宽字符集（Widechars）。Unicode是一种

双字节编码机制的字符集，使用0～65535之间的双字节无符号整数对每个字符进行编码。在Unicode中，所有的字符都是16位，包括所有的7位ASCII码都被扩充为16位（注意，高位扩充的是零）。如字符串“pediy”，它的ASCII码是：

70h 65h 64h 69h 79h

其Unicode码的十六进制是：

0070h 0065h 0064h 0069h 0079h

Intel处理器在内存中，一个字存入存储器要占有相继的两个字节，这个字存放时就按Little-Endian方式存入，即低位字节存入低地址，高位字节存入高地址，如图1.2所示

[image]

图1.2　内存中的Unicode码

1.2　Windows操作系统

本书是研究Windows平台上的加解密，因此要求读者必须对操作系统要有所了解。如有可能，看看Windows操作系统原理方面的书籍，这对以后的一些深入理解是有帮助的。

1.2.1　Win API简介

现在很多讲程序设计的书都是基于MFC库和OWL库的Windows设计，对Windows实现的细节鲜有讨论，而调试程序是和系统底层打交道的，所以很有必要掌握一些API函数的知识。

对于一个初学者来说，API函数也许是一个时常耳闻却感觉有些神秘的东西。API的英文全称为Application Programming Interface（应用程序编程接口）。对这个定义的理解，需要追溯到操作系统的发展历史。当Windows操作系统开始占据主导地位的时候，开发Windows平台下的应用程序成为人们的需要。而在Windows程序设计领域处于发展的初期，Windows程序员所能使用的编程工具唯有API函数。这些函数提供应用程序运行所需要的窗口管理、图形设备接口、内存管理等各项服务功能。这些功能以函数库的形式组织在一起，形成了Windows应用程序编程接口（API），简称Win API。Win API子系统负责将API调用转换成Windows操作系统的系统服务调用，所以，可以认为API函数是构筑整个Windows框架的基石，在它的下面是Windows的操作系统核心，而它的上面则是Windows应用程序，如图1.3所示。对于应用程序开发人员而言，所看到的Windows操作系统实际上就是Win API，操作系统的其他部分对开发人员来说是完全透明的。

[image]

图1.3　Windows应用程序与操作系统的关系

用于16位版本Windows的API（Windows 1.0到Windows 3.1）现在称做Win16。用于32位版本Windows的API（Windows 9x/NT/2000/

XP/2003）现在称作Win32。API函数调用在从Win16到Win32的转变中保持兼容，并在数量和功能上不断增强，从Windows 1.0支持不到450个函数调用，到现在已有几千个函数。

所有32位版本的Windows都支持Win16 API（以确保和旧应用程序兼容）和Win32 API（以运行新应用程序）。非常有趣的是，Windows NT/2000/XP与Windows 9x的工作方式不同。在Windows NT/2000/XP中，Win16函数调用通过一个转换层被转化为Win32函数调用，然后被操作系统处理。在Windows 9x中，该操作正好相反：Win32函数调用通过转换层转换为Win16位函数调用，再由操作系统处理。

Windows运转的核心是一个称做“动态链接”的概念。Windows提供了应用程序可利用的丰富的函数调用，这些函数采用动态链接库即DLL实现。在Windows 9x中通常位于\WINDOWS\SYSTEM子目录中，在Windows NT/2000/XP中通常位于系统安装目录里的\SYSTEM和\SYSTEM32子目录中。

在早期，Windows的主要部分只需要在三个动态链接库中实现。这代表了Windows的三个主要子系统，它们分别叫做Kernel、User和GDI。

• Kernel（由16位的KRNL386.EXE和32位的KERNEL32.DLL实现）：操作系统核心功能服务，包括进程与线程控制、内存管理、文件访问等；

• User（由16位的USER.EXE和32位的USER32.DLL实现）：负责处理用户接口，包括键盘和鼠标输入、窗口和菜单管理等；

• GDI（由16位的GDI.EXE和32位的GDI32.DLL实现）：图形设备接口，允许程序在屏幕和打印机上显示文本和图形。

除了上述模块以外，Windows还提供了其他一些DLL以支持另外一些功能，包括对象安全性、注册表操作（ADVAPI32.DLL）、通用控件（COMCTL32.DLL）、公共对话框（COMDLG32.DLL）、用户界面外壳（SHELL32.DLL）、图形引擎（DIBENG.DLL），以及网络（NETAPI32.DLL）。

[image]注意：Windows 9x是一个16位与32位的混合体，因此系统将Kernel、User和GDI等链接库的16位与32位版本一起加载到系统内存里。Windows NT/2000/XP是一个纯32位操作系统，并没有加载KRNL386.EXE等16位链接库到内存中（也许为了兼容特殊的16位程序，才保留在SYSTEM32目录中）。

Win 32 API是一个基于C语言的接口，但是Win32 API中的函数可以由用不同语言编写的程序调用，只要在调用时遵循调用的规范即可。

1.2.2　常用Win32 API函数

由于Win32程序大量调用系统提供的API函数，而Win32平台上的调试器，如OllyDbg等，恰好有针对API函数设置断点的强大功能，因而掌握常用的API函数具体用法会给跟踪调试程序带来极大的方便。本节将把常用的Win32 API函数介绍一下，详细的Win32 API参考文档可以从MSDN中获得。建议读者掌握一定的Win32编程知识，如阅读《Windows程序设计》一书，这对合理选择API函数有很大的帮助。

API函数是区分字符集的：A表示ANSI；W表示Widechars，即

Unicode。前者就是通常使用的单字节方式，后者是宽字节方式，以方便处理双字节字符。用字符串作参数的每个Win32函数在操作系统中都有两种方式的版本。例如，编程时使用MessageBox函数，而在USER32.DLL中，没有32位MessageBox函数的入口点。实际上，有两个入口点，一个名为MessageBoxA（ANSI版），另一个名为MessageBoxW（宽字符版）。幸运的是，程序员通常不必关心这个问题，代码中只需要使用MessageBox，开发工具中的编译模块就会根据设置决定采用MessageBoxA还是MessageBoxW。

常用API函数详解：

（1）hmemcpy函数

[image]

这是一个Win16 API函数，位于16位的KRNL386.EXE链接库里。但一般的编程书籍上很少提到，原因是它是系统底层的东西，没有特殊需要，一般不直接调用。它执行的操作很简单，只是将内存中的一块数据拷贝到另一个地方。

上文已说过，在Windows 9x中，Win32函数调用通过转换层转换为Win16函数调用，所以Windows 9x底层频繁地调用hmemcpy这个16位的函数来拷贝数据。由于这个特性，它常被解密者作为断点拦截数据，从而有个别称“万能断点”。在Windows NT/2000系统上，相关的函数是memcpy，但在Windows NT/2000上不同于在Windows 9x上，应用程序很少再调用Memcpy来处理数据，用此函数设置断点基本上什么也拦不住。

（2）GetWindowText函数

此函数在USER32.DLL用户模块中，它的作用是取得一个窗体的标题文字，或者一个文本控件的内容。函数原型：

[image]

返回值：如果成功就返回文本长度；失败则返回零值。

ANSI版是GetWindowTextA，Unicode版是GetWindowTextW。

（3）GetDlgItem函数

此函数在USER32.DLL用户模块中，它的作用是获取指定对话框的句柄。函数原型：

[image]

返回值：如果成功就返回对话框的句柄；失败则返回零。

（4）GetDlgItemText函数

此函数在USER32.DLL用户模块中，它的作用是获取对话框文本。函数原型：

[image]

返回值：如果成功就返回文本长度；失败则返回零。

ANSI版是GetDlgItemTextA，Unicode版是GetDlgItemTextW。

（5）GetDlgItemInt函数

此函数在USER32.DLL用户模块中，它的作用是获取对话框整数值。函数原型：

[image]

返回值：如果成功，lpTranslated被设置为TRUE，返回文本对应的整数值；如果失败，lpTranslated被设置为FALSE，返回值为零。

（6）MessageBox函数

此函数是在USER 32.DLL用户模块中，创建和显示信息框。函数原型：

[image]

ANSI版是MessageBoxA，Unicode版是MessageBoxW。

1.2.3　什么是句柄

句柄（Handle）在Windows中使用非常频繁，它是Windows标识，由应用程序建立或使用的对象所使用的一个唯一的整数值（通常为32位）。Windows要使用各种各样的句柄来标识诸如应用程序实例、窗口、图标、菜单、输出设备、文件等对象。程序通过调用Windows函数获取句柄，然后在其他Windows函数中使用这个句柄，以引用它代表的对象。句柄的实际值对程序来说无关紧要，这个值是被Windows模块内部用来引用相应对象的。

当一个进程被初始化时，系统要为它分配一个句柄表，句柄值是放入进程的句柄表中的索引。当调试一个应用程序并且观察内核对象句柄的实际值时，会看到一些较小的值，如1，2等。请记住，句柄的含义并没有记入文档资料，并且可能随时变更。实际上，在Windows 2000中，返回的值用于标识放入进程的句柄表的该对象的字节数，而不是索引号本身。因此，在Windows的不同版本下调试程序时，就不要为句柄值的表达形式不同而疑惑了。

1.2.4　Windows 9x与Unicode

Windows 9x操作系统不是一种全新的操作系统。为了向下兼容，它继承了16位Windows 3.x操作系统的特性。因此，微软并没有把所有的16位函数全部改写成32位的，而是仅仅通过将其包装进32位代码重用了现有的16位代码。这样，这些32位代码会转过来调用16位。在这种Win32 API实现下，KERNEL32的大多数函数都转到了KRNL386，USER32转到了USER.EXE，GDI32转到了GDI.EXE。

如果要增加Windows 9x对Unicode的支持，其工作量相当大。Windows 9x几乎都是使用ANSI字符串来进行所有的内部操作的。但Windows 9x里还是有少量函数具有支持Unicode的能力，如下面的Win9x_Unicode.exe程序：

[image]

在VC编译器中，设置好Unicode标识符，按Unicode方式编译，生成Unicode程序。可以用反汇编工具查看其汇编代码：

[image]

很明显，程序中存在一个Unicode版本的MessageBox函数，这时在Windows 9x上执行这个程序，结果程序正常运行。

这个实例演示了部分Unicode函数能在Windows 9x上运行得很好。现在来看一看Windows 98里MessageBoxW函数的内部结构：

[image]

原来，MessageBoxW函数将Unicode字符串转换成ANSI字符串，最终调用ANSI版的MessageBoxExA函数来显示窗口。

这个实例涉及Unicode与ANSI之间转换字符串的操作，程序调用了WideCharToMultiByte函数进行转换。

如要将ANSI字符串转换成Unicode字符串，可调用MultiByteToWideChar函数来实现。

Windows 9x系统上还有其他几个比较常用的Unicode函数，如lstrlenW，FindResourceW，GetCommandLine，ExtTextOut，TextOutW，MultiByteToWideChar等。

Windows 9x上其他成百上千的函数只提供了接受Unicode参数的进入点，但是这些函数并不将Unicode字符串转换成ANSI字符串，只返回运行失败的消息。这些函数只有ANSI版本才能正确运行。所以，如要为Windows 9x系统开发软件，只能用ANSI版函数开发应用程序，Unicode版本的程序在Windows 9x下无法正常运行。

1.2.5 Windows NT/2000/XP与Unicode

在NT架构上，Win32 API也是以KERNEL32、USER32和GDI32动态链接库提供的。然而，这种实现完全从底层做起没有使用任何的16位代码，因此是Win32 API的纯32位实现。甚至16位代码都最终要调用这些32位API。

Unicode影响到计算机工业的每个部分，对操作系统和编程语言的影响最大。NT系统是使用Unicode标准字符集从头进行开发的，其系统核心完全是用Unicode函数工作的。调用任何一个Windows函数并给它传递一个ANSI字符串，那么系统首先要将字符串转换成Unicode，然后再将Unicode传递给操作系统。相反，如果希望函数返回ANSI字符，系统就会首先将Unicode字符串转换成ANSI字符串，然后将结果返回给应用程序。也就是说，在NT架构下，Win32 API能接受Unicode和ASCII两种字符集，而其内核则只能使用Unicode。所有这些操作对用户来说都是透明的，但进行这些字符串的转换需要占用系统资源。

现在来看一看Windows 2000里MessageBoxA函数的内部结构：

[image]

这个试验结果表明MessageBoxExA函数其实是一个替换翻译层，用于分配内存，并将ANSI字符串转换成Unicode字符串，系统最终是调用Unicode版的MessageBoxExW函数执行。当MessageBoxExW返回时，它便释放内存缓存。在这个过程中，系统必须执行这些额外的转换操作，因此ANSI版的应用程序需要更多的内存和占用更多的CPU资源。而Unicode版的程序在Windows 2000/XP下执行效率就高多了。

Windows 2000/XP既支持Unicode，也支持ANSI，所有新增和未过时的函数在Windows 2000/XP中都同时拥有ANSI和Unicode两个版本。

1.2.6　Windows消息机制

Windows是一个消息（Message）驱动式系统，Windows消息提供应用程序与应用程序之间、应用程序与Windows系统之间进行通信的手段。应用程序想要实现的功能由消息来触发，并且靠对消息的响应和处理来完成。

Windows系统中有两种消息队列：一种是系统消息队列，另一种是应用程序消息队列。计算机的所有输入设备由Windows监控。当一个事件发生时，Windows先将输入的消息放入系统消息队列中，再将输入的消息拷贝到相应的应用程序队列中，应用程序中的消息循环从它的消息队列中检索每个消息并且发送给相应的窗口函数中。一个事件的发生，到达处理它的窗口函数必须经历上述过程。值得注意的是消息的非抢先性，即不论事件的急与缓，总是按到达的先后排队（一些系统消息除外），这就使得一些外部实时事件可能得不到及时的处理。

由于Windows本身是由消息驱动的，所以调试程序时跟踪一个消息会得到相当底层的答案。下面将常用的Windows消息函数列出，以方便需要时参考。

（1）SendMessage函数

调用一个窗口的窗口函数，将一条消息发给那个窗口。除非消息处理完毕，否则该函数不会返回。

[image]

返回值：由具体的消息决定。如消息投递成功，则返回TRUE（非零）。

（2）WM_COMMAND消息

当用户从菜单或按钮中选择一条命令或者一个控件时发送给它的父窗口，或者当一个快捷键被释放时发送。Visual C＋＋的WINUSER.H文件里定义了WM_COMMAND消息对应的十六进制是0111h。

[image]

返回值：如果应用程序处理这条消息，则返回值为零。

（3）WM_DESTROY消息

当一个窗口被销毁时发送。WM_DESTROY消息的十六进制是02h。这条消息无参数。

返回值：如果应用程序处理这条消息，则返回值为零。

（4）WM_GETTEXT消息

应用程序发送一条WM_GETTEXT消息，将一个对应窗口的文本拷贝到一个呼叫程序提供的缓冲区中。WM_GETTEXT消息的十六进制是0Dh。

[image]

返回值：被拷贝的字符数。

（5）WM_QUIT消息

当应用程序调用PostQuitMessage函数时，生成消息WM_QUIT。WM_QUIT消息的十六进制数是012h。

[image]

返回值：这条消息没有返回值。

（6）WM_LBUTTONDOWN消息

当光标在一个窗口的客户区并且用户按下鼠标左键时，WM_LBUTTONDOWN消息被发送。如果鼠标动作未被捕获，这条消息被发送给光标下的窗口；否则，被发送给已经捕获鼠标动作的窗口。WM_LBUTTONDOWN消息的十六进制是0201h。

[image]

返回值：如果应用程序处理这条消息，则返回值为零。

1.3　保护模式简介

学过8088/8086汇编语言的读者，一定熟悉AX，BX，CX，DX，SI，DI，SP，BP和IP这些16位的寄存器；在80386中，这些寄存器被扩展到了32位，即EAX，EBX，ECX，EDX，ESI，EDI，ESP，EBP，EIP，段寄存器增加了两个：FS和GS。

在64位CPU中，这些寄存器被扩展到了64位，即RAX，RBX，RCX，RDX，RSI，RDI，RSP，RBP，RIP，新增8个通用寄存器（R8～R15）。所有的这些寄存器能够具有字节、字、双字和四字4个级别。

一般来说，80x86（80386及其以后的各代CPU）可在实模式、保护模式和虚拟86模式3种模式下运转。实模式就是古老的MS-DOS的运行环境，只能利用这些32位寄存器的前16位，而后面的16位就浪费了。当前流行的Windows操作系统运行在保护模式下，在保护模式下程序可以利用更多的内存，可以实现多任务系统。

1.3.1　虚拟内存

在保护模式下，CPU的寻址方式与实模式不同。实模式下的寻址方式是“段基址＋段偏移”，段的默认大小为64KB，所有段都是可读/写的，唯有代码段是可执行的，段的特权级为0。而在保护模式下内存是“线性”的，因为这时段寄存器的意义不同，它里面存放的不再是段基地址，而是存放着段选择子，这个值是不直接参与寻址的，只是全局描

述符表（Global Descriptor Table, GDT）或本地描述符表（Local Descriptor Table, LDT）的一个指针，不同段寄存器有不同的属性（读、写、执行、特权级等），如图1.4所示。尽管如此，在继续看保护模式内存结构时，仍请记住段/偏移量的概念，不妨把段寄存器看做是对保护模式中选择子的一个模拟。关于段描述符的定义，读者可参考其他保护模式的书籍资料。

[image]

图1.4　保护模式的寻址操作

Win32的平坦内存模式使每个进程拥有赋予它自己的虚拟空间，对于32位进程来说，这个地址空间是4GB，因为32位指针可以拥有从00000000h ~ FFFFFFFFh之间的任何一个值。此时，程序的代码和数据都放在同一地址空间中，即不必区分代码段和数据段。程序员也不必了解段寄存器CS，DS，ES等的具体内容。

虚拟内存（Virtual Memory）不是真正的内存，它通过映射（Map）的方法，使可用的虚拟地址（Virtual Address）达到4GB，每个应用程序可以被分配2GB的虚拟地址，剩下的2GB留给操作系统自己用。在Windows NT中，应用程序甚至可有3GB的虚拟地址。Windows是一个分时的多任务操作系统，CPU时间被分成一个个的时间片后分配给不同程序，在一个时间片里，和这个程序执行无关的东西并不映射到线性地址中。因此每个程序都有自己的4GB寻址空间，互不干扰。在物理内存中，操作系统和系统DLL代码需要供每个应用程序调用，所以在所有的时间必须映射；用户EXE程序只在自己所属的时间片内被映射，而用户DLL则有选择地被映射。

简单地说，虚拟内存的实现方法和过程如下：

① 当一个应用程序被启动时，操作系统就创建一个新进程，并给每个进程分配2GB的虚拟地址（不是内存，只是地址）；

② 虚拟内存管理器将应用程序的代码映射到那个应用程序的虚拟地址中的某个位置，并把当前所需要的代码读取到物理地址中（注意：虚拟地址和应用程序代码在物理内存中的位置是没有关系的）；

③ 如果使用动态链接库DLL，DLL也被映射到进程的虚拟地址空间，在需要的时候才被读入物理内存；

④ 其他项目（例如数据、堆栈等）的空间是从物理内存中分配的，并被映射到虚拟地址空间中；

⑤ 应用程序通过使用它的虚拟地址空间中的地址开始执行，然后虚拟内存管理器把每次的内存访问映射到物理位置。

如果看不明白上面的步骤也不要紧，但要明白以下几点：

• 应用程序是不会直接访问物理地址的；

• 虚拟内存管理器通过虚拟地址的访问请求，控制所有的物理地址访问；

• 每个应用程序都有相互独立的4GB寻址空间，不同应用程序的地址空间是隔离的；

• DLL程序没有自己的“私有”空间，它们总是被映射到其他应用程序

的地址空间中，作为其他应用程序的一部分运行。因为如果它不和其他程序同属一个地址空间，应用程序就无法调用它。

使用虚拟内存的好处是：简化了内存的管理，并可弥补物理内存的不足；可以防止多任务环境下各个应用程序之间的冲突。

1.3.2　保护模式的权限级别

在保护模式下，所有的应用程序都有权限级别（Privilege Level, PL），这个权限级别按优先次序分为4等：0，1，2和3，其中3特权级别最低，0特权级别最高。特权级环如图1.5所示。

[image]

图1.5　特权级环

如果应用程序拥有第0级的权限（也就是说，其PL＝0，或者它是运行在Ring 0的应用程序），它就可以执行所有的指令并访问所有数据；如果应用程序拥有的权限级别是第3级，它能执行的指令是有限的，能访问的数据也是有限的。操作系统核心层是运行在Ring 0级的，而Win32子系统（如动态链接库KERNEL32.DLL、USER32.DLL和GDI32.DLL）是运行在Ring 3级的，以提供与应用程序的接口。

图1.6所示是Windows 2000/XP体系结构简图，核心态工作在Ring 0级，用户态工作在Ring 3级。HAL是一个可加载的核心模块HAL.DLL，它为运行在Windows 2000/XP上的硬件平台提供低级接口。Windows 2000/XP的执行体是NTOSKRNL.EXE的上层（内核是其下层）。用户层导出并且可以调用的函数接口在NTDLL.DLL中，通过Win32 API或一些其他的环境子系统对它们进行访问。

[image]

图1.6　Windows 2000/XP体系结构简图

另外，用户的应用程序也是运行在Ring 3级的（就是用Visual C＋＋，Borland C＋＋，Visual Basic，Delphi，Borland C＋＋ Builder等SDK工具开发的应用程序），也就是说，享有的权限是最低的——换言之，受到保护模式的“保护”。该类应用程序没有权限去破坏操作系统，只能规矩地使用Win32 API接口函数与系统打交道。如想控制系统，就必须取得0特权级，比如调试工具SoftICE就是工作在0特权级上的。

1.4　认识PE格式

Windows的可执行文件（EXE, DLL）是PE（Portable Executable）格式。本节先简单介绍一下PE文件，详细的PE格式请参考第10章“PE文件格式”。

PE文件使用的是一个平面地址空间，所有代码和数据都被合并在一起，组成一个很大的结构。文件的内容被分割为不同的区块（Section，又称区段、节等），块中包含代码或数据。每个块都有它自己在内存中的一套属性，比如：这个块是否包含代码、是否只读或可读/写等。

每一个区块都有不同的名字，这个名字用来表示区块的功能。例如，一个叫.rdata的区块表明它是一个只读区块。常见的块有.text，.rdata，.data，.idata，.rsrc等。各种块的含义如下。

• .text：是在编译或汇编结束时产生的一种块，它的内容全是指令代码；

• .rdata：是运行期只读数据；

• .data：是初始化的数据块；

• .idata：包含其他外来DLL的函数及数据信息，即输入表；

• .rsrc：包含模块的全部资源，如图标、菜单、位图等。

PE文件非常好的一个地方就是在磁盘上的数据结构与在内存中的结构是一致的，如图1.7所示。装载一个可执行文件到内存中，主要就是将一个PE文件的某一部分映射到地址空间中。这样，PE文件的数据结构在磁盘和内存中是一样的。

[image]

图1.7　PE文件结构

PE相关名词解释如下。

（1）入口点（Entry Point）

PE文件执行时的入口点（Entry Point）。也就是说，程序在执行时的第一行代码的地址应该就是这个值。

（2）文件偏移地址（File Offset）

当PE文件储存在磁盘上时，各数据的地址称做文件偏移地址（File Offset）。文件偏移地址从PE文件的第一个字节开始计数，起始值为0。

（3）虚拟地址（Virtual Address, VA）

由于Windows程序运行在386保护模式下，所以程序访问存储器所使用的逻辑地址称为虚拟地址（Virtual Address, VA），又称为内存偏移地址（Memory Offset）。与实地址模式下的分段地址类似，虚拟地址也可写成“段:偏移量”的形式，这里的段是指段选择子。例如，“0167:00401000”就是这种表示方法，其中：

• 0167：这是段选择子，其数据保存在CS段选择器里。同一程序在不同系统环境下，此值可能不同，一般也不需要关心此值。

• 00401000：此处表示内存的虚拟地址（Virtual Address），一般来说，同一程序的同一条指令在不同系统环境下，此值相同。

（4）基地址（ImageBase）

文件执行时将被映射到指定内存地址中，这个初始内存地址称为基地址（ImageBase）。这个值是由PE文件本身设定的。按照默认设置，用Visual C＋＋建立的EXE文件基地址是00400000h，DLL文件基地址是10000000h。但是，可以在创建应用程序的EXE文件时改变这个地址，方法是在链接应用时使用链接程序的/BASE选项。

用PE编辑工具（如LordPE）可以查看可执行的PE字段。单击LordPE的“PE Editor”打开任意一个可执行文件，如图1.8所示。

[image]

图1.8　LordPE中的PE编辑器

LordPE功能非常强大，其显示的各项内容，学习第10章后就能理解。若要查看区块的信息，单击“Sections”按钮来打开区块编辑器（见图1.9）。

[image]

图1.9　区块（Section）数据

在图1.9中显示出可执行文件在磁盘与内存中各区块的地址、大小等信息。虚拟地址和虚拟大小是指该区块在内存中的地址和大小。物理地址和物理大小是指该区块在磁盘文件中的地址和大小。

第2篇　调试篇

- 第2章　动态分析技术
- 第3章　静态分析技术
- 第4章　逆向分析技术

调试逆向是软件安全技术的基础。本篇以极大的篇幅，以动态分析与静态分析技术为主线，讲解了代码的逆向分析技巧，同时介绍了逆向工程必备工具OllyDbg、SoftICE、IDA的操作技巧等。

第2章　动态分析技术

动态分析技术中最重要的工具是调试器，分为用户模式和内核模式两种类型。用户模式调试器是指用来调试用户模式应用程序的调试器，它们工作在Ring 3级，如OllyDbg、Visual C＋＋等编译器自带的调试器。内核模式调试器是指能调试操作系统内核的调试器，它们处于CPU和操作系统之间，工作在Ring 0级，如SoftICE等。

2.1　OllyDbg 调试器

OllyDbg（简称OD）是由Oleh Yuschuk（www.ollydbg.de）编写的一款具有可视化界面的用户模式调试器，可以在当前各种Windows版本上运行，但NT的系统架构更能发挥OllyDbg强大功能。OllyDbg结合了动态调试和静态分析，具有GUI界面，非常容易上手，并且对异常的跟踪处理相当灵活，这些特性使得OllyDbg成为调试Ring 3级程序的首选工具。它的反汇编引擎很强大，可识别数千个被C和Windows频繁使用的函数，并能将其参数注释出。它会自动分析函数过程、循环语句、代码中的字符串等。此外，开放式的设计给了这个软件很强的生命力，爱好者不断地修改、扩充OllyDbg，脚本执行能力和开放插件接口使得其变得越来越强大。

2.1.1　OllyDbg界面

本节以OllyDbg 1.10讲述其用法，支持32位程序。OllyDbg发行版本是一个ZIP压缩包，只要将其解压缩到一个目录下，然后运行ollydbg.exe即可。打开目标程序后，OllyDbg会打开多个子窗口，单击菜单View或单击工具栏标签L、E、M等按钮可在各子窗口间切换，如图2.1所示。这些按钮依次对应Log窗口、Executable modules窗口、Memory窗口、Threads窗口、Windows窗口、Handles窗口、CPU窗口、Patches窗口、Call stack窗口、Breakpoints窗口、References窗口、Run trace窗口、Source窗口等。更多窗口请查看View菜单，各窗口功能请参考OllyDbg帮助文档的描述。

[image]

图2.1　窗口切换面板

默认的当前窗口是CPU窗口，它在OllyDbg中是最重要的窗口。调试程序的绝大部分操作都要在这个窗口中进行。它包括以下5个面板窗口：反汇编面板、寄存器面板、信息面板、数据面板、堆栈面板，如图2.2所示。

[image]

图2.2　OllyDbg主界面

各窗口的外观属性，如标题栏（bar）、字体（font）等在右键菜单“Appearance”（界面选项）里控制。

1．反汇编面板窗口（Disassembler window）

反汇编面板窗口显示被调试程序的代码。它有4个列：地址（Address）、机器码（Hex dump）、反汇编代码（Disassembly）和注释（Comment）。最后一列注释栏显示相关API参数或运行简表，非常有用。

在反汇编面板窗口的列中（注：不是列标题），默认时，双击完成以下动作。

- Address列：显示相对被单击地址的地址，再次双击返回到标准地址模式；
- Hex dump列：设置或取消无条件断点，对应的快捷键是F2键；
- Disassembly列：调用汇编器，可直接修改汇编代码；
- Comment列：允许增加或编辑注释，对应的快捷键是“;”键。

要从键盘上选择多行，按下Shift键和上下光标箭头或者PgUp/PgDn键，也可利用右键菜单命令。按Ctrl键并按上下光标箭，一行一行地滚动汇编窗口（当数据与代码混合时，此功能非常有用）。

2．信息面板窗口（Information window）

动态跟踪时，显示与指令相关的各寄存器值、API函数调用提示和跳转提示等信息。

3．数据面板窗口（Dump window）

以十六进制和字符方式显示文件在内存中的数据。要显示数据，可单击鼠标右键“Go to expression”命令或按“Ctrl + G”键打开地址窗口，输入地址。

4．寄存器面板窗口（Registers window）

显示CPU各寄存器的值，支持浮点、MMX和3DNow!寄存器，可以单击鼠标右键切换或单击窗口标题切换显示寄存器的方式。

5．堆栈面板窗口（Stack window）

显示了堆栈的内容，即ESP指向地址的内容。堆栈窗口非常重要，各API函数和子程序等都利用它传递参数和变量等。

2.1.2　OllyDbg的配置

OllyDbg的设置在菜单Options里，有界面选项（Appearance）和调试选项（Debugging options）等。这些选项配置都保存在ollydbg.ini文件里。

1．界面设置

单击菜单“Options/Appearance”打开界面选项对话框，单击“Directories”（目录）标签，这里设置UDD文件和插件的路径，为了避免可能出现的问题，请设置成绝对路径，如图2.3所示。

[image]

图2.3　UDD文件及插件路径设置

UDD文件是OllyDbg的工程文件，用以保存当前调试的一些状态，如断点、注释等，以便下次调试时继续使用。

插件用以扩充功能，路径设置正确后，将插件复制到这个目录，在OllyDbg的主菜单里就会出现“Plugin”（插件）这个菜单项。

OllyDbg界面外观完全可以定制，这些外观由Appearance选项里的Fonts，Colours，Code highlighting控制。

颜色（Colours）：这里是OllyDbg颜色的主题，可以根据自己的喜欢设置。设置时，先选择合适的颜色主题（Colour scheme），例如本例是“Black on white”，如图2.4所示，然后进行配色。主题配置好后，就可在其他窗口应用这个主题，如在CPU窗口单击右键，选择菜单“Appearance/Colours/Black on white”，这样新的配置才会生效。

[image]

图2.4　OllyDbg界面颜色设置

代码高亮显示（Code highlighting）：此选项主要作用于CPU窗口，可设置相关代码的颜色，提高代码可读性。操作与颜色设置一样。

2．调试设置

单击菜单“Options/Debugging options”打开调试设置选项对话框，一般按默认设置即可。其中异常（Exceptions），可以设置让OllyDbg忽略或不忽略那些异常，现在建议全部选上，如图2.5所示。有关异常的知识后面章节会讲解。

[image]

图2.5　调试选项中的异常设置

3．加载符号文件

这个功能类似IDA的FLIRT，使用符号库（Lib），可以让OllyDbg以函数名显示DLL中的函数。例如MFC42.DLL是以序号输出函数的，这时在OllyDbg显示的是序号，如果让其加载MFC42.DLL调试符号，则以函数名显示相关输出函数。加载方法是单击菜单“Debug/Select import libraries”来打开导入库窗口，如图2.6所示。

[image]

图2.6　加载调试符号库

4．关联到右键菜单

可以将OllyDbg关联到资源管理器右键菜单里，调试程序时，只需要在EXE或DLL文件上单击右键，就会出现“Open with Ollydbg”菜单。

要实现关联只需要单击菜单“Options/Add to Explorer”，再单击“Add OllyDbg to menu in Windows Explorer”按钮即可关联。

2.1.3　加载程序

OllyDbg可以用两种方式加载目标程序调试，一种是通过CreateProcess创建进程；另一种是利用DebugActiveProcess函数将调试器捆绑到一个正在运行的进程上。

1．利用CreateProcess创建进程

单击菜单“File/Open”或按快捷键F3打开目标文件，这样会调用CreateProcess创建一个用以调试的新进程。OllyDbg将接收到目标进程发生的调试事件，而对其子进程的调试事件将不予理睬。

OllyDbg除了直接加载目标程序外，也支持带参数的程序，方法是：在打开对话框中的“Arguments”栏中输入参数行，如图2.7所示。

[image]

图2.7　带参数调试程序

2．将OllyDbg附加到一个正在运行的进程上

OllyDbg的一个实用的功能是可以调试正在运行的程序，这个功能称为“附加（Attach）”。其原理是利用DebugActiveProcess函数可以将调试器捆绑到一个正在运行的进程上，如果执行成功，则效果类似于利用CreateProcess创建的新进程。

单击菜单“File/Attach”打开附加对话框，如图2.8所示。选中正在运行的目标进程，单击Attach按钮即可附加目标进程。附加后，目标程序会暂停在Ntdll.dll的DbgBreakPoint处，在OllyDbg里按一下F9键或Shift＋F9键让程序继续运行。接着就可对目标程序进行调试分析了。

[image]

图2.8　附加目标进程

[image]注意：附加一个程序时，尽量用新打开的OllyDbg，这样附加的成功率高些。

如果是隐藏进程，就不能用上述方法附加了。OllyDbg有一个-p启动参数，只要得到进程的pid就可以附加了。用IceSword等工具获得隐藏进程的pid，然后在控制台窗口用-p参数附加即可。注意，pid的值是十进制。

[image]

如果附加不成功，可以巧妙利用OllyDbg的即时调试器功能来调试。先看一个例子，运行A.exe，其会调用B.exe，此时用OllyDbg附加B.exe，OllyDbg会无响应。解决办法：在“Options/Just-in-time

debugging”中设置OllyDbg为即时调试器，将B.exe的入口改成CC，即INT 3指令，同时记下原指令。运行A.exe，其调用B.exe，运行到INT 3指令会导致异常，OllyDbg会作为即时调试器启动并加载B.exe，此时再将INT 3指令恢复原指令，继续调试。

2.1.4　基本操作

对于习惯Borland开发环境的朋友来说，用OllyDbg比较容易上手，例如单步功能是用F7键和F8键等，这与Borland的产品习惯完全一样。

在这里，以一个用Visual C++ 6.0编译的程序TraceMe来讲解OllyDbg的操作，编译时优化选项按默认设置为“Maximize speed”。读者也可以按“Minimize Size”优化选项编译一下，并与本文比较。因为优化选项不同，生成的汇编代码也会有所不同。

1．准备工作

拆解一个Windows程序要比拆解一个DOS程序容易得多，因为在Windows中，只要API函数被使用，想对寻找蛛丝马迹的人隐藏一些东西是比较困难的。因此分析一个程序，用什么API函数作为切入点就显得比较关键了，如果有些编程经验，这方面就更得心应手了。

为了便于理解，先简单地看一下TraceMe的序列号验证流程，如图2.9所示。将姓名与序列号输入到文字框中，程序调用GetDlgItemTextA函数把字符读出来，然后进行计算，最后用函数lstrcmp进行比较。因此，这些调用的函数就是解密跟踪的目标，用这些函数作为断点，跟踪程序的序列号验证过程就能找出正确的序列号。

[image]

图2.9　TraceMe程序序列号验证过程

2．加载目标文件调试

为了能让OllyDbg中断在程序的入口点，加载程序前必须要设置一下。运行OllyDbg后，单击菜单“Options/Debugging options”，打开调试选项配置对话框，再单击“Event”标签，如图2.10所示。这里设置OllyDbg对中断入口点、模块加载／卸载、线程创建／结束等事件的处理，一般调试只需要将暂停点设置在“Entry point of main module”或“WinMain”即可。

[image]

图2.10　设置OllyDbg第一次暂停

• System breakpoint：系统断点，OllyDbg用CreateProcessA加载DEBUG_ONLY_THIS_PROCESS参数执行，程序运行之后会触发一个INT 3，在系统空间里。

• Entry point of main module：主模块的入口点，即文件的入口点。

• WinMain：程序的WinMain()函数入口点，即使设置这个选项，

OllyDbg一般也只会中断在文件入口点处。

设置好后，单击菜单“File/Open”打开TraceMe.exe，此时OllyDbg会中断在TraceMe的入口点，如图2.11所示。光条停在4013A0这行，这个4013A0就是程序开始执行的入口地址（EntryPoint）。

[image]

图2.11　OllyDbg加载目标程序停在入口点

在图2.11中，代码中各部分含义如下：

• ① 虚拟地址：一般情况下，同一程序的同一条指令在不同系统环境下此值相同；

• ② 机器码：这就是CPU执行的机器代码；

• ③ 汇编指令：和机器码对应的程序代码。

3．单步跟踪

调试器的一个最基本功能就是动态跟踪，OllyDbg在菜单“Debug”里控制运行的命令，各个菜单项都有相应的快捷键。OllyDbg的单步跟踪功能键如表2-1所示。

表**2-1　OllyDbg**的单步跟踪功能键

[image]

F8键在调试中用得很频繁，可以一句句地单步执行汇编指令，遇到CALL指令不会跟进，而路过。例如：

[image]

F7和F8功能键的主要差别就在于若遇到CALL、LOOP等指令，F8键是路过，而F7键是跟进去。

[image]

当要重复按多次F7键或F8键时，OllyDbg提供了“Ctrl＋F7”和“Ctrl＋F8”快捷键，直到用户按Esc键、F12键或遇到其他断点时停止。

当位于某个CALL中，这时想返回到调用这个CALL的地方时，可以按“Ctrl＋F9”快捷键执行“执行到返回（Execute till return）”功能。OllyDbg就会停在遇到的第一个返回命令（RET、RETF或者IRET），这样可以很方便地略过一些没用的代码。例如上面的代码，在401DA0这行，如果按“Ctrl＋F9”快捷键就会返回到4013FF这句。遇到RET指令是暂停还是路过可以在选项里设置，方法是：打开调试设置选项对话框，在“Trace”页面，设置“After Execting till RET, step over RET（执行到RET后，单步路过RET）”。

如果跟进系统DLL提供的API函数中，此时想返回到应用程序领空里，可以按快捷键“Alt＋F9”执行“Execute till user code（执行到用户代码）”命令。例如：

[image]

在上面的4013C6一行，按F7键就可跟进系统KERNEL32.DLL里的领空：

[image]

像地址7C8114AB等都是系统DLL所在的地址空间，这时只要按一下快捷键“Alt + F9”就可回到应用程序领空里。代码如下：

[image]

[image]注意：所谓领空，实际上是指在某一时刻，CPU的CS:EIP所指向的某段代码的所有者。

如果不想单步跟踪，让程序直接运行起来，可以按F9键或单击工具栏中的[image]按钮。如果想重新调试目标程序，可以按“Ctrl + F2”快捷键或单击工具栏中的[image]按钮，Ollydbg结束被调试进程并重新加载它。有时程序进入死循环，可以按F12键暂停程序。

4．设置断点

断点是调试器的一个重要功能，可以让程序中断在需要的地方，从而方便对其分析。最常用的断点是INT 3断点，其原理是Ollydbg将断点地址处的代码修改为INT 3指令。在图2.12中，将光标移动到4013A5一行，按F2键即可设置一个断点，再按一次F2键取消断点。也可以用鼠标双击“Hex dump”列中相应的行设置断点，再次双击取消断点。当关闭程序时，OllyDbg会自动将当前应用程序的断点位置保存在其安装目录*.udd文件中，以便下次运行时，这些断点继续有效。如果将断点设置到当前应用程序代码外，OllyDbg将会警告。可以在菜单“Options/Debugging options/Security”选项里将“Warn when breakpoint is outside the code section”取消选中，以关闭这个警告。

[image]

图2.12　设置断点

现在开始一个完整的调试分析过程，取消开始设置的所有断点，在OllyDbg里按F9键将实例TraceMe.exe运行起来，如图2.13所示。

[image]

图2.13　实例运行起来的界面

字符通常利用Windows文本框输入。为了检查输入的字符，程序常采用下面这些函数把文本框中的内容读出来，如表2-2所示。

表**2-2**　程序采用的将文本框中内容读出来的函数

[image]

一般事先不会知道程序具体是调用了什么函数来处理字符的，只好多试几遍，找出相关的函数。

首先，需要在OllyDbg中设定一个“陷阱”（或称断点）。因为这个TraceMe是32位ANSI版的程序，所以在GetDlgItemTextA处设一个断点。按“Ctrl + G”键打开跟随表达式的窗口，输入GetDlgItemTextA字符，如图2.14所示。

[image]

图2.14　打开跟随表达式窗口

[image]注意：OllyDbg里对API的大小写敏感，输入的函数名大小写必须正确。

单击OK按钮后，会来到系统USER32.DLL中的GetDlgItemTextA函数入口处，如图2.15所示。

[image]

图2.15　跳到函数入口处

在77D6AC1E这一行，按F2键设个断点，即在GetDlgItemTextA函数入口处设了断点（操作系统版本不同，这个函数入口地址是不一样的），如果这个函数被调用，OllyDbg就会中断。

[image]注意：在Windows 9x系统中，OllyDbg是无法对API函数入口点下断的，因此不能用此方法设断。

下一步可以列出所有断点来检查一下，按“Alt + B”快捷键或单击[image]按钮打开断点窗口，如图2.16所示。

[image]

图2.16　断点窗口

这里可以显示除硬件断点外的其他断点，其中“Always”表示断点处于激活状态，“Disable”表示断点停用，按空格键可切换其状态，也可以用鼠标右键菜单管理这些断点。

现已经设定了断点，可以捕捉任何对GetDlgItemTextA函数的调用。然后输入姓名和序列号，如姓名为“pediy”，序列号为“1212”。单击“Check”按钮，程序中断在OllyDbg中，就在函数GetDlgItemTextA开始的地方。

也可通过输入表设置断点。在OllyDbg里，按“Ctrl + N”键打开应用程序的输入表，会发现USER32.GetDlgItemTextA函数，在这个函数上按Enter键或右键菜单执行“Find references to import”命令打开调用此函数的参考代码窗口，找到相应的代码，按Enter键即可切换到相应的代码，接下来按F2键设置断点。

5．调试分析

按“Alt + F9”键，回到调用函数的地方，当然也可以按F8键单步走出GetDlgItemTextA这个函数。OllyDbg非常强大，已将各函数的调用参数及当前值都注释出来了。相关代码如下：

[image]

来到TraceMe领空后，可以按“Alt + B”键打开断点窗口，将GetDlgItemTextA处的断点禁止。

很多时候必须重复跟踪同一段代码，因此可以先设置一个断点。将光标移到4011AE一行，按F2键设置新的断点，以方便反复跟踪调试。

中断后的代码如下（可结合源码阅读）：

[image]

[image]

在阅读这些代码时：

• 要搞清各API函数的定义（查看相关API手册）。

• API函数基本采用的是__stdcall调用约定，即函数入口参数按从右到左的顺序入栈，并由被调用者清理栈中参数，返回值放在eax寄存器中。因此，对相关的API函数要分析其前的push指令，这些指令将参数放进堆栈以传送给API调用。整个跟踪过程中要关注堆栈数据变化。

• C代码中的子程序采用的是C调用约定，函数入口参数按从右到左的顺序入栈，由调用者清理栈中的参数。

有关调用约定、参数传递等知识，可以从本书第4章获得。阅读上面代码时，需理解的GetDlgItemTextA函数原型如下：

[image]

GetDlgItemText采用标准调用约定，参数按从右到左的顺序入栈。例如本例中的汇编代码：

[image]

当GetWindowText函数执行后，将把取出的文本放到由lpString（LPTSTR是一个长的指针，指向由空字符终止的字符串）指定的位置。如想看到输入的字符串，跟踪的时候，在4011B0一行停住，在eax寄存器单击右键，执行菜单“Follow in Dump”命令查看数据窗口中的内容，当然此时数据窗口中没什么有价值的东西。继续按F8键单步执行完下面一句：

[image]

此时GetDlgItemTextA函数已将字符串取出，放到eax所指的地址里。数据窗口右边字符段显示出刚输入的字符“pediy”，如图2.17所示。

[image]

图2.17　数据窗口查看字符

6．保存修改后的文件

在上一节中，已找到序列号的判断核心，这里的一段代码是关键：

[image]

只要4011F5一句不跳转即可注册成功。在调试过程中，当执行到4011F5一句时，有几种方法可以验证判断。

• 在OllyDbg寄存器面板，用鼠标单击标志寄存器ZF（即“Z”），双击一次ZF的值取反，如果原来是1，执行后为0，如图2.18所示。

[image]

图2.18　改变标志寄存器的值

• 在4011F5一行，双击鼠标或单击空格键，输入指令：“NOP”，这个指令机器码是90，如图2.19所示，此处用“9090”取代“7437”。

[image]

图2.19　键入汇编代码

修改好后的代码如下：

[image]

现在随意输入姓名与序列号，这个CrackMe都会提示注册成功。目前修改的是内存中数据，为了使修改一直有效，就必须将这个变化写进磁盘文件中去。OllyDbg也提供了这个功能，方法是用鼠标选中修改过的代码，单击鼠标右键，执行“Copy to executable/Selection”命令，如图2.20所示。

[image]

图2.20　保存修改的文件

执行复制到可执行文件的命令后，将打开文件编辑窗口，如图2.21所示。单击鼠标右键，执行命令“Save File”即可将修改保存到文件。像这种屏蔽程序的某些功能或改变程序流程，使程序的保护方式失效的方法称为patch（补丁）或“爆破”。

[image]

图2.21　文件编辑器

7．算法分析

接下来，分析序列号算法的原理，通常这个过程比较复杂。序列号核心计算部分的高级语言（C语言）形式如下：

[image]

下面代码是调用GenRegCode()函数，在4011E5一行，按F7键进入这个函数，同时，注意堆栈窗口的数据变化。

[image]

进入子程序call 00401340后，子程序初始化堆栈，此时堆栈情况如图2.22所示。

[image]

图2.22　堆栈面板窗口

程序在此利用esp来访问各参数，ebp用来处理字符串name[i]。详细过程如下：

[image]

[image]

计算序列号用的数据表可从这句指令中查到：

[image]

停在这一句，来到数据窗口，按“Ctrl + G”键输入地址：405030，查看数据窗口，如图2.23所示。

[image]

图2.23　查看数据窗口

数据窗口显示的就是Table表，其值为：0C 0A 13 09 0C 0B 0A 08。

TraceMe最后调用了函数lstrcmp来比较字符，它的原型是：

[image]

调用代码如下：

[image]

因此执行到40138F一句时，堆栈窗口中就会显示出正确的序列号2470。如图2.24所示，左边是数据窗口显示的数据，右边是堆栈窗口，直接将指向的字符串显示出来了。

[image]

图2.24　数据窗口查看序列号字符

这样一个程序就分析完了，读者感兴趣，可以写出这段代码逆算法，写出注册机。

2.1.5　断点

常用的断点有INT 3断点、硬件断点、内存断点等。调试时，合理使用断点，能大大提高效率。

1．INT 3断点

当执行一个INT 3断点时，该地址处的内容被调试器用INT 3指令替换了，此时OllyDbg将INT 3隐藏了，显示出来仍是下断前的指令，如图2.12按F2键下的断点。实际上，4013A5处的指令68已被替换成CC了：

[image]

这个INT 3指令，其机器码是CCh，也常称为CC指令。当被调试进程执行INT 3指令导致一个异常时，调试器就会捕捉这个异常从而停在断点处，然后将断点处的指令恢复成原来指令。当然，如果自己写调试器，也可用其他一些指令代替INT 3来触发异常。

用INT 3断点的好处是可以设置无数个断点，缺点是改变了原程序指令，容易被软件检测到。例如为了防范API被下断，一些软件会检测API的首地址是否为CCh，以此来判断是否被下了断点。在这用C语言来实现这个检测，方法是取得检测函数的地址，然后读取它的第一个字节，判断它是否等于“CCh”。下面这段代码就是对MessageBoxA函数进行的断点检测：

[image]

程序编译后，对MessageBoxA设断，程序将会发现自己被设断跟踪。当然躲过检测的方法是将断点下在函数内部或末尾，例如可以将断点下在函数入口的下一行，就可躲过检测了。

2．硬件断点

硬件断点和DRx调试寄存器有关。从Intel CPU体系架构手册中，可以找到DRx调试寄存器的介绍，如图2.25所示。

[image]

图2.25　Intel调试寄存器示意图

DRx调试寄存器总共有8个，从DR0到DR7。每个寄存器的特性如下：

• DR0 ~ DR3：调试地址寄存器，保存需要监视的地址，如设置硬件断点；

- DR4 ~ DR5：保留，未公开具体作用；
- DR6：调试寄存器组状态寄存器；
- DR7：调试寄存器组控制寄存器。

硬件断点原理是使用4个调试寄存器（DR0,DR1,DR2,DR3）来设定地址，以及DR7设定状态，因此最多只能设置4个断点。

OllyDbg支持调试寄存器，其称为硬件断点。设断方法是在指定的代码行单击鼠标右键，执行“Breakpoint/Hardware, on execution（断点／硬件执行）”命令。

为了便于理解，这里演示一下。加载实例TraceMe.exe，右键单击寄存器面板窗口，执行“View debug registers（查看调试寄存器）”，接着在4013AA这行设置硬件断点。按F9键执行程序，程序就会中断在4013AA这一行，查看调试寄存器，会发现DR0的值为4013AA，如图2.26所示。

[image]

图2.26　演示硬件断点

设置断点后，OllyDbg实际上就是将DR0 ~ DR3其中的一个设置为4013AA，然后在DR7中设定相应的控制位。这样当被调试进程运行到4013AA时，CPU就会给OllyDbg发送异常信息，OllyDbg将该信息做初步处理后，中断下来，让用户继续进行操作。

硬件断点删除稍有些麻烦，单击菜单“Debug/Hardware breakpoints（调试／硬件断点）”，打开硬件断点面板，如图2.27所示，然后单击“Delete”按钮删除相应的硬件断点。

[image]

图2.27　删除硬件断点

OllyDbg提供了一个快捷键F4，可以执行到光标所在的行，也是利用调试寄存器原理，中断后自动删除，相当于一次性硬件断点。

硬件断点优点是速度快，在INT 3断点容易被发现的地方，使用硬件断点来代替会有很好的效果；缺点就是最多能使用4个断点。

3．内存断点

OllyDbg可以设置内存访问断点或内存写入断点，原理是对所设的地址设为不可访问／不可写属性，这样当访问／写入的时候就会产生异常，OllyDbg截获异常后比较异常地址是不是断点地址，如果是就中断，让用户继续进行操作。

内存断点会降低OllyDbg速度，因为每次异常时都要通过比较来确定是否应该停下，也许OllyDbg可能在速度上做了考虑而只实现一个内存断点。

程序运行时会有3种状态：读取、写入、执行。

[image]

用OllyDbg加载实例TraceMe.exe，看到4013D0一行有一个写内存的指令：

[image]

就用这个地址来演示一下如何下内存断点。在数据窗口对405528下内存写断点，方法是将光标移到405528地址处，选中需要下断点的地址区域，单击鼠标右键，执行“Breakpoint/Memory, on write（断点/内存写入）”，如图2.28所示。

[image]

图2.28　设置内存写入断点

下了内存写断点后，按F9键让程序跑起来，会马上中断在“4013D0 mov[405528], edx”这行。如果要清除内存断点，单击鼠标右键，执行“Breakpoint/Remove memory breakpoint（断点/删除内存断点）”。同样，内存访问断点操作类似。

对代码也可下内存访问断点。在OllyDbg里重新加载实例，随意定位一行代码，如4013D6，单击鼠标右键，执行“Breakpoint/Memory, on access（断点/内存访问）”，如图2.29所示。

[image]

图2.29　设置内存访问断点

当然要执行内存地址4013D6的代码时需要“访问”它，因此按F9键让实例在OllyDbg里跑起来，就会中断在4013D6这行所下的内存访问断点上。这个实验表明内存执行的地方，也可以用内存访问中断。

内存断点不修改原代码。它不会像INT 3断点那样，因为修改代码被程序校验而导致中断失败，因此在遇到代码校验，并且硬件断点失灵情况下，可以用内存断点来代替。

4．内存访问一次性断点

Windows对内存使用段页式的管理，在OllyDbg里按“Alt + M”键显示内存，可以看到许多段，每个段都有不可访问、读、写、执行属性。在相应的段上单击右键，如图2.30所示，会发现一个命令“Set break-on-access（在访问上设置断点）”，其快捷键是F2键，对整个内存块设置该类断点。这个断点是一次性断点，当所在段被读取或执行时就中断，中断发生以后，断点将被删除。想捕捉调用或返回到某个模块时，如后面章节中的脱壳时，该类断点就显得特别有用。右键中的“Set memory breakpoint on access（设置内存访问断点）”和“Set break-on-access”功能一样，所不同的是它不是一次性断点。这类断点仅在NT架构下可用。

[image]

图2.30　对区块设置内存断点

5．消息断点

Windows本身是由消息驱动的，如果调试时没有合适的断点，可以尝试消息断点。消息断点使得当某个特定窗口函数接收到某个特定消息时程序中断。

当用户单击一个按钮、移动鼠标或向文本框中键入文字时，一条消息就会被发送给当前的窗体。所有发送的消息都有4个参数：一个窗口句柄（hwnd），一个消息编号（msg），还有两个32位长度（Long）的参数。Windows通过句柄来标识它代表的对象，比如单击某个按钮，Windows就是通过句柄来判断是点击了哪一个按钮，然后发送相应的消息通知程序。

用实例TraceMe.exe来演示一下如何下消息断点。在OllyDbg里运行实例，输入用户名与序列号，单击菜单“View/Windows（查看／窗口）”或单击工具栏中的[image]按钮，列出窗口相关参数，如图2.31所示。如果界面无内容显示，此时执行鼠标右键菜单中的“Actualize（刷新）”命令。

[image]

图2.31　列出窗口相关参数

这里用于列出所有属于被调试程序窗口及其窗口相关的重要参数，比如按钮、对应的ID以及句柄（Handle）等。现在要对Check按钮下断点，当单击按钮时中断。在Check条目上单击鼠标右键，如图2.32所示。

[image]

图2.32　设置消息断点

在弹出的右键菜单中，执行“Message breakpoint on ClassProc（在ClassProc上设置消息断点）”，会弹出如图2.33所示的设置窗口。

[image]

图2.33　在WinProc上设消息断点

当用鼠标左键单击按钮并松开时，会发送WM_LBUTTONUP这个消息，单击图2.33中的下拉菜单选择“202 WM_LBUTTONUP”，再单击“OK”按钮，至此消息断点设置好了。

回到TraceMe界面，单击“Check”按钮，鼠标松开时，将会中断在Windows系统代码里。代码如下（不同版本的系统，代码会不同的）：

[image]

现在消息捕捉到了，但处于系统底层代码里，这时企图使用“Alt + F9”键或“Ctrl + F9”键返回到TraceMe程序的领空代码里是徒劳的。

VC可执行文件的执行代码是存放在代码段里的，本例就是.text区块

里。当从系统代码回到应用程序代码段的时候，正是代码段的执行，因此对代码段下内存断点就能返回应用程序的代码领空。按“Alt + M”键打开内存窗口，对.text区块下内存访问断点，执行右键菜单命令“Set break-on-access（在访问上设置断点）”或按快捷键F2，如图2.34所示。

[image]

图2.34　对代码段下内存访问断点

现在按F9键运行程序，立即中断在程序的空间004010D0处，这里正是程序的消息循环处。

[image]

这段代码是一个消息循环，不停地处理TraceMe主界面的各类消息，此时可能不是直接处理按钮事件，如果单步跟踪，会跟进系统代码里去。在系统代码里，再次按“Alt + M”键打开内存窗口，对.text区块下内存访问断点，按F9键运行后，会再次来到代码里。可以重复这个过程，在1、2次中断后，就能到达处理按钮的事件代码。“Check”按钮事件的代码：

[image]

最后，可以将消息断点删除，方法是按“Alt + B”键切换到断点窗口，选中消息断点，直接删除，如图2.35所示。

[image]

图2.35　删除消息断点

6．条件断点

在调试过程中，经常希望断点满足一定条件时才中断，这类断点称为条件断点。OllyDbg的条件断点可以按寄存器、存储器、消息等设断。条件断点是一个带有条件表达式的普通INT 3断点。当调试器遇到这类断点时，它将计算表达式的值，如果结果非零或者表达式有效，则断点生效（即暂停被调试程序）。有关条件表达式的规则描述请参考OllyDbg帮助文档。

（1）按寄存器条件中断

用OllyDbg打开光盘映像文件中实例Conditional_bp.exe，在401476这行，按条件断点的快捷键“Shift + F2”，如图2.36所示。

[image]

图2.36　下条件断点

在条件框内，输入条件表达式“eax＝＝0400000”。这样，程序执行

到401476这行时，如果eax值为400000h，OllyDbg将中断。如果安装了命令行插件，也可在命令行里直接输入：

[image]

（2）按存储器条件中断

在这以CreateFileA函数演示一下。在实际情况中，程序可能成百上千次地调用CreateFileA函数，让OllyDbg在CreateFileA打开所需要文件时中断，显得十分有必要。先来看一下CreateFile函数的定义：

[image]

运行实例Conditional_bp，对CreateFileA设断，单击“OpenTest”按钮，如图2.37所示是当OllyDbg中断时从堆栈中看到的。图中最左侧标识了各参数相对于当前ESP的地址，打开这个功能的方法是在堆栈窗口单击右键，执行“Address/Relative to ESP（地址／相对于ESP）”菜单。

[image]

图2.37　CreateFileA参数刚入堆栈的情形

CreateFile采用标准调用约定，参数按从右到左的顺序入栈。因为在函数刚执行时，EBP堆栈结构还未建立，只得用ESP访问这些参数。CreateFile函数第一个参数FileName是文件名指针，在OllyDbg里如要得到第一个参数的内存地址，可以用[ESP＋4]来获得；如果还要得到此处地址所指的字符串，必须用[[ESP＋4]]来表示。实例Conditional_bp调用了4次CreateFileA函数，假设当CreateFile打开“c:\\1212.txt”时需要OllyDbg中断，条件断点可以这样设置，将光标移到CreateFileA函数第一行，按快捷键“Shift＋F2”，键入字符“[STRING [esp＋4]]＝＝"c:\\1212.txt"”，STRING前缀在OllyDbg中的解释是以零作为结尾的ASCII字符串，如图2.38所示。

[image]

图2.38　条件断点

如果安装了命令行插件，也可以直接输入：

[image]

7．条件记录断点

条件记录断点除了具有条件断点作用，还能记录断点处函数表达式或参数的值。也可以设置通过断点的次数，每次符合暂停条件时，计数器减一。

例如要记录Conditional_bp实例调用CreateFileA函数的情况，在CreateFileA函数第一行，按“Shift＋F4”键，出现条件记录窗口，如图2.39所示。

[image]

图2.39　设置条件记录断点

在Condition（条件）域中输入要设置的条件表达式。Explanation（说明）域中由用户自己设置一个名称。Expression（表达式）域中是要记录的内容的条件，只能设置一个表达式，例如要记录EAX的值，可以输入EAX。还有一个Decode value of expression as（解码表达式的值）下拉选择框，这是对记录的数据进行分析，例如在图2.39中，如果Expression域中填的是[esp＋4]，则得在下拉选择框中选“Pointer to ASCII String（指向ASCII字符串的指针）”，才能得到正确的结果，功能相当于STRING前缀。

Pause program（暂停程序）是指OllyDbg遇到断点时是否中断；Log value of expression（记录表达式值）是指遇到断点时是否记录表达式的值；Log function arguments（记录函数参数）是指遇到断点时是否记录函数参数。可以根据需要设置Never（从不）、On condition（按条件）、Always（永远）等条件。

条件记录断点允许传递一个或多个命令给插件。当应用程序因条件断点暂停，并且断点包含有传递给插件的命令时，都会调用回调函数ODBG_Plugincmd(int reason,t_reg*registers,char*cmd)。例如，当程序暂停时，传送一个命令“d esp”给CmdBar插件，只要在图2.39所示的框中输入字符“.d esp”，注意命令前有一个点字符“.”，那么当条件断点断下时，就会执行“d esp”这个命令，这时我们就可以在数据窗口中看到ESP地址处的数据了。

设置好条件记录断点后，单击实例Conditional_bp的“OpenTest”按钮，运行后，OllyDbg会在Log data窗口（快捷键“Alt＋L”）记录下数据，如图2.40所示。

[image]

图2.40　查看Log data窗口

2.1.6　插件

OllyDbg支持插件，这意味它的功能扩展性很好，可以按自己需要扩展相关的功能，提高调试的灵活性。首先按图2.3设置插件所在的目录，将相应的插件复制到这个目录下，重新运行OllyDbg就可加载插件。OllyDbg默认只能加载32个插件，另外，各插件之间有可能会冲突，因此建议插件目录仅放常用的插件，以减少各类问题出现。

在此介绍一下自带的命令行插件。

1．常用插件介绍

光盘映像文件中的OllyDbg已集成了常用的插件，这些插件使用都比较简单，本节不过多介绍，读者可参考相关的资料。

（1）命令行插件CmdBar

单击菜单“Plugins/command line/command line”打开命令行插

件，如图2.41所示。

[image]

图2.41　加载命令行插件的效果

命令参数较多，详细信息请参考其自带的帮助文件，在此列出几个常用命令，如表2-3所示。

表**2-3　CmdBar**插件常用命令

[image]

（2）OllyScript插件

OllyDbg的脚本插件，可以用OllyScript脚本来完成一些复杂的操作或重复的操作，具体使用可参考其ReadMe。

2．插件的开发

开发插件，到OllyDbg官方站点下载文档OllyDbg Plugin API v1.10及SDK，具体方法可以参考光盘映像文件中提供的样例。

2.1.7　Run trace

Run trace（Run跟踪）可以把被调试程序执行过的指令保存下来，了解以前发生的事件。它能将地址、寄存器的内容、消息等记录到Run trace缓冲区中。在运行Run trace前，要将缓冲区设置大些，否则执行的指令太多造成缓冲区溢出，这时OllyDbg会自动丢弃老的记录。可以在“Debugging options/Trace（调试选项／跟踪）”面板中设置，如图2.42所示。

[image]

图2.42　设置Run trace缓冲区

如果要将Run trace的数据保存到文件，在跟踪之前，单击菜单“View/Run trace”或[image]按钮，打开Run trace窗口，单击右键执行“Log to file（记录到文件）”，如图2.43所示。

[image]

图2.43　设置Run trace缓冲区

需要运行Run trace时，单击菜单“Debug/Open or clear run trace（打开或清除Run跟踪）”。在打开Run trace缓冲区后，OllyDbg会记录执行过程中的所有暂停，然后通过使用“＋”键和“-”键（“＋”键必须是数字键盘上的）来反方向浏览程序的执行，此时OllyDbg会使用实际的内存状态来解释寄存器、堆栈的变化。

在反汇编窗口显示的是被调试程序领空时，在反汇编窗口的快捷菜

单中选择“Run trace/Add entries of all procedures（Run trace／添加所有函数过程的入口）”，这样能够检查每个可识别的函数被调用的次数，如图2.44所示。运行后，可以在Run trace窗口的快捷菜单中执行“Profile module（统计模块）”查看统计次数。

[image]

图2.44　添加Run trace选项

按F9键或者按“Ctrl＋F12”组合键（跟踪路过）让程序运行，查看Run trace结果只需单击菜单“View/Run trace”或[image]按钮，打开Run trace窗口。

2.1.8　Hit trace

Hit trace能够让调试者辨别哪一部分代码执行了，哪一部分没有。OllyDbg的实现方法相当简单，它将选中区域的每一条命令处均设置一个INT 3断点，当中断发生的时候，OllyDbg便把它去除掉。在使用Hit trace的时候，不能在数据中设置断点，否则程序可能会崩溃。

当碰到一段跳转分支比较多的代码时，需要了解程序执行线路，可以用Hit trace。方法是选中这段代码，单击右键执行“Hit trace/Add selection”，将需要监视的代码选中，然后按F9键让程序运行，OllyDbg会在已被执行过的指令前用另一种颜色标记出来。

[image]注意：如果右键菜单中没有Hit trace，则必须打开相关的菜单选项，进行代码分析。比如可以按“Ctrl＋A”键或执行快捷键中“Analysis/Analyse code（分析／分析代码）”命令重新分析一下代码。

2.1.9　符号调试技术

高级语言编译器都附带源代码级的调试器，如Visual C＋＋、Delphi等。OllyDbg等调试器除了进行汇编级调试外，也可在源代码状态下调试应用程序，当然要实行源码调试需要符号信息。

1．符号格式

符号表（又称调试符）的作用是将十六进制数转换为源文件代码行、函数名以及变量名称。符号表还包含程序使用的类型信息，调试器使用类型信息可以获取原始数据，并将这些原始数据显示为在程序中所定义的结构或变量。

（1）SYM格式

SYM格式早期用于MS-DOS和16位Windows系统，现在只用做Windows 9x的调试符（因为Windows 9x多数内核仍然是16位代码）。

（2）COFF格式

COFF格式（Common Object File Format）是UNIX供应商所遵循的规范的一部分。Windows NT 2.1首次引进使用，现在微软逐渐抛弃COFF格式，而使用更流行的符号表达式。

（3）C7格式（Code View格式）

最早是在MS-DOS作为Microsoft C/C＋＋ 7的一部分而问世的，现

在已经支持Win32系统。Code View是早期Microsoft调试器的名称，其支持的调试符号为C7格式。C7格式在执行模块中是自我包含的，符号信息与二进制代码混合，进而意味着调试文件会非常大。

（4）PDB格式

PDB（Program Database）格式是现今最常用的一种符号格式。Visual C＋＋和Visual Basic都支持PDB格式。PDB与C7不同，PDB符号根据应用程序不同的链接方式，保存在单独的一个文件或多个文件中。

（5）DBG格式

DBG是系统调试符号，有了系统调试符号，调试器才可以显示系统函数名。DBG文件与其他符号格式不同，因为链接器并不创建DBG文件，DBG文件基本是一个包含其他调试符号的文件，如包含COFF或C7等类型调试符号。微软将操作系统调试符都分配在DBG文件中，当然这些文件只包括公用信息和全局信息，如ntdll.dbg、kernel32.dbg等。Windows 9x没有系统调试符，Windows NT/2000/XP提供了系统调试符。

（6）MAP文件

MAP文件是程序的全局符号、源文件和代码行号信息的唯一文本表示方法。MAP文件在任何地方、任何时候都可以使用，不要求支持程序，通用性极好。

2．创建调试文件

进行源码级调试的首要条件是生成的文件包含调试信息。调试信息包含程序里的每个变量的类型和在可执行文件里的地址映射以及源代码的行号。调试器利用这些信息使源代码和机器码相关联。各语言编译器都能产生相关调试信息，具体见表2-4。

表**2-4**　编译器设置调试信息

[image]

在这以Visual C 6.0为例，建立带有PDB调试信息的调试版本：

① 单击“Build”菜单，单击“Set Active Configuration”配置，从对话框中选择“Win32 Debug”配置。

② 在“Project”菜单中单击“Settings”打开设置对话框，单击“C/C＋＋”标签，从“Category”下拉框中选择“General”，在调试信息（Debug Info）部分，选择“Program Database”选项。该选项产生一个存储程序信息的数据文件（.PDB），它包含了类型信息和符号化的调试信息。

③ 单击“Link”标签，从“Category”下拉框中选择“Debug”，在调试信息（Debug Info）部分，选择“Debug info”、“Microsoft format”、“Separate types”，在此也可选择“Generate mapfile”来生成MAP文件。

3．用符号文件调试

用Visual C 6.0编译光盘映像文件中提供的TraceMe源码，运行OllyDbg打开Debug目录下带有调试信息的TraceMe，这次显示的汇编代

码带有调试符，具有更好的可读性。

[image]

在OllyDbg主菜单中单击“View/Source”打开源码窗口，可看到源码。OllyDbg不能直接在源码窗口单步调试，此时必须做些调整，用CPU窗口来配合调试。同时打开CPU窗口和Source窗口，并适当地调整它们的位置和大小。然后在“Debugging options”里设置“CPU”标签，勾选“Synchronize source with CPU”，这样跟踪时汇编与源码就能同步了。也可以在CPU窗口中，单击第4栏的标题，直接在CPU窗口里同步显示汇编和源码。

可单击菜单“View/Source files”打开源码文件路径窗口查看，一些路径不正确的源文件，OllyDbg默认将不显示。需要显示，可以在“Debugging options”里设置，切换到“Debug”标签，将“Hide non-existing source files”选中去除，即可显示不存在的源文件路径。

2.1.10 OllyDbg常见问题

OllyDbg这款调试器非常强大，初学者刚接触可能会遇到许多问题，在这将一些常见的问题列出。

1．乱码的问题

当用OllyDbg跟踪程序时，可能会出现如下情况：

[image]

这是因为OllyDbg将这段代码当成数据了，没有进行反汇编识别。在右键快捷菜单中，执行“Analysis/Analyse code（分析／分析代码）”或按“Ctrl + A”键强迫OllyDbg重新分析一下代码即可。如果还不行，尝试在快捷菜单中执行“Analysis/Remove analysis from module（分析／从模块中删除分析）”或在UDD目录中删除相应的.udd文件。

调整后就以代码显示了：

[image]

2．如何快速回到当前领空

在OllyDbg中有时查看代码可能翻页到其他地方，如想快速回到当前CPU所在的指令上，可以双击寄存器面板中的EIP或单击[image]按钮。

3．OllyDbg如何修改**EIP**

首先将光标移到所需要的地址上，然后执行右键菜单“New origin here（此处新建EIP）”或使用快捷键“Ctrl + *”。

4．什么是**UDD**

OllyDbg把所有程序或模块相关的信息保存至单独的文件中，并在

模块重新加载时继续使用。这些信息包括了标签、注释、断点、监视、分析数据、条件等。

5．为什么删除了断点，**OllyDbg**重新加载时，这些断点都会重新出现

设置ollydbg.ini，在配置文件里改成：Backup UDD files＝1

6．**OllyDbg**反汇编窗口键入汇编代码时，输入**“push E000”**会提示未知标识符（见图**2.45**）

[image]

图2.45　添加Run trace选项

这是因为OllyDbg的反汇编引擎不能正确识别字符“E000”中的E是字母还是数字。解决的办法是，在字母前加个0，表示这是数字，即“push 0E000”。

7．**OllyDbg**出会现假死现像

用OllyDbg调试一些加壳程序，程序跑到断点（包括硬件断点）时，OllyDbg会出现假死现象。解决方法是打开配置文件ollydbg.ini，如果“Restore windows”是一个很大的值，现在只需要设“Restore windows＝0”即可。

8．如何微调窗口显示

可以通过“Ctrl + ↑”键或“Ctrl + ↓”键对反汇编窗口或数据窗口翻动一个字节。

9．执行复制到可执行文件时， 出错**“Unable to locate data in executable file”**

这个要修改的地方不在RawSize范围内，修改PE文件，使“RawSize＝VirtualSize”。

10．能否把**CALL**调用改成函数名形式

比如“call 401496”，假设401496处是amsg_exit函数，将光标停在401496，单击“Shift + ;”键，出来一个标签框，输入字符“amsg_exit”，这样，所有调用401496的CALL都变成“call < amsg_exit > ”形式。

2.2　SoftICE调试器

SoftICE是一款优秀的系统级调试工具。Compuware公司已在2006年4月宣布DriverStudio停止开发，这意味着SoftICE将在新系统中消失。本节只简单介绍一下SoftICE基本用法，详细的请参考其自带的文档Using SoftICE.pdf和CommRef.chm。

本书实例用OllyDbg都可调试，当遇到Ring 0级程序时，才需要SoftICE、WinDBG，因此读者可以跳过此节的学习。

由于篇幅限制，本节内容以电子文档形式放在光盘映像文件中提

供。

第3章　静态分析技术

用高级语言编写的程序有两种形式，一种被编译成机器语言在CPU上执行，如Visual C＋＋、Pascal等。由于机器语言与汇编语言几乎是对应的，因此可将机器语言转化成汇编语言，这个过程称为反汇编（Disassembler）。例如，在x86系统中，机器码“EBh”对应的汇编语句是“jmp short xx”。另一种高级语言是一边解释一边执行的，称为解释性语言，如Visual Basic 3.0/4.0、Visual FoxPro等，这类语言编译后的程序可以被还原成高级语言的原始结构，这个过程称为反编译（Decompiler）。

所谓静态分析，即从反汇编、反编译手段获得程序汇编代码或源代码，然后从程序清单上分析程序流程，了解模块完成的功能。

3.1　文件类型分析

静态分析的第一步就是分析程序的类型，了解程序是用什么语言编写的或用什么编译器编译的？程序是否被某种加密程序处理过？然后才能有的放矢地进行下一步工作。

常见的文件分析工具有PEiD、FileInfo等，本节简单地讲一讲它们的用法。

3.1.1　PEiD工具

PEiD是一款常用的文件检测分析工具，具有GUI界面。它能检测大多数编译语言、病毒和加密的壳。图3.1显示被分析的文件是用Microsoft Visual C＋＋ 6.0编译的，分析不出类型的文件可能报告是“PE Win GUI”，Win GUI就是Windows图形用户界面程序的统称。使用时，建议设置一下Options，勾上“Register Shell Extensions”，即可在鼠标右键添上快捷菜单。

[image]

图3.1　PEiD主界面

PEiD这类文件分析工具是利用查特征串搜索来完成识别工作的。各种开发语言都有固定的启动代码部分，利用这点就可识别出是何种语言编译的。被加壳程序处理过的程序，在壳里会留下相关加壳软件的信息，利用这点就可识别是被何种壳所加密的。

PEiD提供了一个扩展接口文件userdb.txt，用户可以自定义一些特征码，这样就可识别出新的文件类型。签名的制作可以用插件Add Signature来完成，必要时，还必须用OllyDbg等调试器来配合修正。有些外壳程序为了欺骗PEiD等文件识别软件，会将一些加壳信息去除，并伪造启动代码部分。例如，将入口代码改成与Visual C＋＋ 6.0所编程程序入口处类似代码，即可达到欺骗目的。所以，文件识别工具所给出的结果只是个参考，文件是否被加壳处理过，还得跟踪分析程序代码才可

得知。

3.1.2　FileInfo工具

FileInfo（简称Fi）是另一款不错的文件检测工具。Fi运行时是DOS界面，在DOS窗口中运行程序相当不便，建议采用下面的技巧：

• 第一种方法是用鼠标将文件拖到Fi.exe主文件上。

• 第二种方法是先建Fi的一个快捷方式，然后把这个快捷方式放进Windows的SendTo文件夹里（Winodws XP用户需将“隐藏文件和文件夹”功能关闭，才能看到SendTo文件夹）。以后要分析某文件，只需选中文件，然后右键单击，选择“发送到”功能就可打开Fi（见图3.2）。建议读者把一些常用工具放进SendTo文件夹，以方便操作。

[image]

图3.2　SendTo菜单

FileInfo与PEiD相比有更强的文件识别能力，也就是说，更难欺骗些。但FileInfo的识别库不能自定义，并且升级也缓慢；而PEiD用户可自定义特征码，因此识别的壳的类型更多。

3.2　静态反汇编

本节主要介绍常见的反汇编工具用法。在进行反汇编前，建议用FileInfo、PEiD等检测工具分析一下文件是否加壳。如果加壳，就需利用后面章节介绍的脱壳技术脱壳，然后再反汇编。常用的反汇编工具有W32Dasm、C32asm、IDA Pro等。W32Dasm、C32asm这两款工具已不升级，不支持64位程序。W32Dasm使用比较简单，其操作方法以电子文档形式放到光盘映像文件中提供。C32asm这款工具十分优秀，其字符串提取功能等十分强大，并且将十六进制工具等功能结合了起来。IDA Pro是一款商业软件，属于专家级产品，功能强大但操作也复杂，逆向工程必备工具。一些简单的程序，可以用W32Dasm、C32asm等来分析，比较方便。

3.2.1　反汇编引擎

反汇编引擎的作用是把机器码解析成可以汇编指令，开发反汇编引擎需要对Intel公司的i386的机器指令编码有深入的了解。不过，一般不需要自己开发，网上有开源或收费的反汇编引擎，利用这些反汇编引擎，自己也可开发一款反汇编分析工具。

常见的反汇编引擎有udis86，ade，xde等，像mlde32和virxasm反汇编引擎也比较有特色，大小仅400字节左右。OllyDbg自带的反汇编引擎也比较强大，但其指令集不全，对MMX，SSE支持的不好，不过FullDisasm插件可以解决这个问题。

3.2.2　IDA Pro简介

IDA Pro（简称IDA）是DataRescue公司（www.datarescue.com）出品的一款交互式反汇编工具，它功能强大、操作复杂，要完全掌握

它，需要很多知识。IDA最主要的特性是交互和多处理器。操作者可以通过对IDA的交互来指导IDA更好地反汇编，IDA并不自动解决程序中的问题，但它会按用户的指令找到可疑之处，用户的工作是通知IDA怎样去做。比如人工指定编译器类型，对变量名、结构定义、数组等定义等。这样的交互能力在反汇编大型软件时显得尤为重要。多处理器特点是指IDA支持常见处理器平台上的软件产品。

IDA支持的文件类型非常丰富，除了常见的PE格式，还支持Windows，DOS，UNIX，Mac，Java，.NET等平台的文件格式。单击菜单“File/Open”打开目标软件ReverseMe.exe，IDA一般能自动识别出格式，如图3.3所示。也可单击菜单“File/New”以向导模式打开文件。

[image]

图3.3　IDA打开文件

IDA打开文件分自动和手工两种模式，默认是自动模式，手工模式（Manual load）可以自由选择需要加载的区块。IDA装载PE文件时，是按区块来装载的，比如.text（代码块）、.data（数据块）、.rsrc（资源块）、.idata（输入表）、.edata（输出表）等。IDA反汇编的时间与程序大小及复杂程度有关，需要等一段时间才能完成。此过程分两个阶段，第一阶段将程序的代码和数据分开，然后为各函数作标记并分析其参数调用，分析跳转、调用等指令关系并给标签赋值等。在第二阶段，如果IDA识别出文件的编译类型，就装载对应的编译器特征文件，然后给各函数赋名。

Kernel option1、Kernel option2、Processor option这三个选项可以控制反汇编引擎的工作状态，一般按默认即可。IDA会自动识别程序类别与处理器类型，在大多数情况下，分析选项的默认值在准确性与方便性之间提供一个折中参数，如果IDA分析出有问题的代码时，将核心选项1中的“Make final analysis pass”选项关闭是一个很好的方法。在有些情况下，一些代码因不在预计的位置而不被确认，尝试将核心选项2中的“Coagulate Data Segments in the final pass”选上是有帮助的。

IDA打开文件后，默认是处于图形化模式下，如果要切换到代码模式，执行右键菜单“Text view”。

3.2.3　IDA的配置

合理配置IDA文件，可以大大提高工作效率。Windows图形界面的主程序是idag.exe，可通过菜单“Options（选项）”来配置IDA，但是这种配置仅对当前的项目有效，新建一个项目时，会恢复成默认配置，改变默认配置必须编辑ida.cfg文件。

在IDA的cfg目录下查找ida.cfg或idagui.cfg文件，这是一个文本文件。不能用Windows记事本打开ida.cfg编辑，因为记事本对一些特殊字符识别不好，继续编辑和保存文件，文件将会被破坏。此时配置文件会导致IDA运行失败，如图3.4所示。建议用户用EditPlus、UltraEdit等工具修改ida.cfg等配置文件。

[image]

图3.4　ida.cfg文件损坏警告窗口

ida.cfg文件由两部分组成。第一部分定义文件的扩展名、内存、屏幕等设置；第二部分配置普通参数，如代码显示格式、ASCII字符串显示格式、脚本定义和处理器选项等。另外，一些问题也与ida.cfg有关，如MAX_ITEM_LINES，默认为5000行，对于许多大文件经常会不够使用，因而发生错误。

1．反汇编选项（Disassembly）

这个选项直接控制反汇编窗口的代码显示格式。单击菜单“Options（选项）”中的“General（常规）”，将出现如图3.5所示的对话框。

[image]

图3.5　反汇编选项配置

ida.cfg与这部分配置对应的是文本格式（Text representation）。由于其项目较多，下面只列出重点部分。配置如下：

[image]

2．ASCII字符串与符号（ASCII strings&names）

ASCII字符串风格可在菜单“Options/ASCII String styles”中打开字符串设置窗口，对应的ida.cfg部分配置如下：

[image]

3．显示中文字符

IDA默认是不显示中文字符串的，只需要在ida.cfg中搜索AsciiStringChars，将cp866 version这段注释掉，恢复full version这段即可显示中文。

[image]

3.2.4　IDA主窗口界面

IDA分析完目标程序后进入主窗口，界面显得专业和复杂。IDA相当智能，尽量分析程序各模块的功能，并给出相应提示，例如为API函数的参数自动加上注释，相当直观易懂。对于那些IDA不能正常分析的代码，则需要手工来辅助分析。

1．翻页

当执行跳转功能后，需要返回时，只要在工具栏中单击[image]按钮或按Esc键，列表便会往后跳一页；若要往前一页，单击[image]按钮

或按“Ctrl＋Enter”键，有点像浏览器。

2．子窗口

在工具栏上单击[image]按钮或从菜单“View/Open subviews/Disassembly”中打开反汇编子窗口，这样可以用多个子窗口来分析同一段程序，不必来回翻页查看代码了。

3．导航器

单击菜单“View/Toolbars/Navigation”打开导航器，如图3.6所示。各部分含义：Library function为库函数，Regular function为规则函数，Instruction为指令，Data为数据，Unexplored为未查过的，External symbol为外部符号，这样根据需要可快速跳到相关代码部分。

[image]

图3.6　导航器

在导航器中执行右键菜单“Zoom in”、“Zoom out”可以调整导航条的显示比例。对于手工分析，导航器的作用非常大，选择适当的倍率可以起到意想不到的效果。

4．注释

IDA可以方便地在代码后面输入注释。在窗口右边空白处单击鼠标右键，显示输入注释的菜单，一种是Enter comment（快捷键是冒号），另一种是Enter repeatable comment（快捷键是分号）。按分号键输入的注释，所有交叉参考处都会出现，按冒号键输入的注释只在该处出现。如果一个地址有两种注释，则只显示非重复注释。

5．提示窗口

IDA最下面的提示窗口主要反馈各种信息。若同时安装了IDA的高低不同版本，有可能导致该窗口消失。解决办法是打开注册表，查找“Datarescue”并删除该主键即可。

3.2.5　交叉参考

通过交叉参考（XREF）可以知道指令代码相互调用的关系。如图3.7所示，这句“CODE XREF:sub_401120＋B↑j”，表示该调用地址是401120，“j”表示跳转（jump）。其他一些符号：“o”表示偏移值（offset），“p”表示子程序（procedure）。双击这里或按回车键可以跳到调用该处的地方。

[image]

图3.7　交叉参考

在loc_401165字符上按X键，将打开交叉参考窗口，如图3.8所示。

[image]

图3.8　交叉参考窗口

3.2.6　参考重命名

“参考重命名（Renaming of reference）”是IDA的一个极好功能，增加了代码的可读性。如图3.9所示的这段代码是窗口函数WndClass的开始处，IDA默认用“loc_401120”命名，但“loc_401120”这个字符没有多大意义。

[image]

图3.9　重命名参考点

若加注解，只有这一行才有意义。但用参考重命名功能便可把所有参考点一次改动。在“loc_401120”上单击鼠标右键，弹出右键菜单，在菜单上选择重命名“Rename”，或按N键，打开“Rename Address”对话框，如图3.10所示。

[image]

图3.10　重命名对话框

在此处给它赋予“WndProc”这个有意义的名字。单击“OK”按钮后马上可以看到所有“loc_401120”标签都改变为新名称，如图3.11所示。

[image]

图3.11　改名后的参考点

3.2.7　标签的用法

在菜单“Jump/Mark position”中打开“标记当前位置”功能，出现如图3.12所示的对话框。

[image]

图3.12　标记当前位置

为这个标记（当前光标位置）加上标签，“WndProc”标签便是需要返回的位置；当离开这个标记而返回时，在菜单“Jump/Jump to marked position”中或按“Ctrl + M”键执行“跳到标记位置”功能（见图3.13）。选择返回的标签，双击就转到指定代码处。

[image]

图3.13　选择标签对话框

3.2.8　进制的转换

IDA可以提供多种进制显示。先将光标移到需要转换的数据上，单

击工具栏上的[image]按钮（见图3.14），即可转换为所需要的进制。“Toggle leading zeroes”功能是用0填补数据前的空位。

[image]

图3.14　进制转换

3.2.9　代码和数据转换

很多工具在反汇编的时候可能无法正确区分数据和代码，IDA也不例外。有些程序就是利用这点来对抗静态反汇编的。IDA的交互性使得用户可以将某段十六进制数指定为代码或数据，即利用人脑来区分代码和数据。

如果确信某段十六进制数据是一段指令，只要将光标移到其第一个字节的偏移位置，执行菜单命令“Edit/Code”或按C键。按P键可以将某段代码定义为子程序，参数调用会列出。若要取消定义，执行菜单命令“Edit/Undefine”或按U键，数据重新以十六进制数据显示。这种交互式分析功能的介入，令IDA获得非交互式软件所不能达到的效果。

在代码行按D键，数据类型会在db、dw与dd间转换。当然可以设置更多的数据类型，执行菜单“Options/Setup data types”命令就可以设置，如图3.15所示。

[image]

图3.15　设置数据类型选项

如果一个字节已经被转换过，再次转换，IDA将会提示让你确认，如图3.16所示。如果感觉这种提示比较麻烦，可以在“Options/Misc”菜单“Convert already defined bytes”命令里关闭。

[image]

图3.16　数据转换提示确认框

3.2.10　字符串

编程语言的不同造成字符串也有不同的格式，如以“0”结尾的C字符串，以“$”结尾的DOS字符串等，IDA支持所有的格式。如果确信某段十六进制数据是一个字符串，只要将光标移到其第一个字符的偏移位置，执行菜单命令“Edit/Strings/ASCII”或在工具栏上单击[image]按钮或按A键。单击[image]按钮右边的小三角，会弹出IDA所支持的字符串格式，如图3.17所示。

[image]

图3.17　选择字符串类型

按A键默认是C字符串，也可以在菜单“Options/ASCII string

style”中设置其他字符串格式为默认值，如图3.18所示。

[image]

图3.18　设置默认字符串格式

IDA有时无法确定ASCII字符串，这种错误的发生是由于这个字符串在程序中没有直接被调用到。本例中，按字母G，输入地址40478E，会来到如下一段代码处：

[image]

将光标移到40478E行并按A键，这句字符串就被定义并生成一个变量名，如要恢复，则按U键。IDA会在生成的字符变量前面加个前缀“a”，如“aGetfiletype db 'GetFileType',0”。

可以在Names窗口看到这些字符串变量，只要单击[image]按钮或从菜单“View/Open subviews/Names”中打开这个窗口。

3.2.11　数组

在处理数据时，可以按数组形式显示。先将光标移到需要处理的数据处，选择菜单“Edit/Array”或单击[image]按钮或按*号键，打开数组排列调整窗口（见图3.19），此处数据按4×12格式排列，见图3.20。其中，若在“Items on a line”（每行项数）中填0，则每行项数根据页面自动调整；若想每行显示更多数据，可以在反汇编选项中调整右边距（Right margin）。

[image]

图3.19　数组排列调整

[image]

图3.20　数据按4×12排列

3.2.12　结构体

IDA会根据文件的类型自动加载相应的类型库，如vc6win（Visual C＋＋ 6.0），用户做底层分析时，可以增加mssdk（windows.h）、ntddk（ntddk.h）等。这些类型库中会有相应的结构体，用户分析代码时，可以直接引用。在工具栏上单击[image]按钮（或按快捷键“Shift＋F11”），打开加载类型库窗口（Loaded Type Libraries），如图3.21所示。用鼠标右键选择“Load Type Library”（或按快捷键Insert），在弹出的窗口中选择类型库，如图3.22所示。

[image]

图3.21　加载类型库窗口

[image]

图3.22　选择类型库

选择好类型库，就可查看内置的结构体数据结构了。选择菜单“View/Open subviews/Structures”或单击工具栏上的[image]按钮，打开结构体管理窗口。按Insert键，在弹出的窗口中单击“Add Standard Structure”，打开添加标准结构库窗口，查找需要的结构名，然后就可以正常使用这些库了。

在默认情况下，IDA会加载常用的结构。对于IDA 5.x版本，在结构体管理窗口里按Insert键，再单击Cancel按钮，ReverseMe程序内常用的结构体数据结构会显示出来。在WNDCLASSA结构一行按“＋”键展开结构，程序代码的相应代码处直接以结构体形式表示，如图3.23所示。

[image]

图3.23　查看结构体

用户也可自定义结构体，例如下面这段C程序：

[image]

如图3.24所示的代码是IDA在反汇编时由于没有定义结构体而自动生成的。

[image]

图3.24　定义结构体前的代码

[esi＋18h]等是调用结构体中的数据，在这可以用有含义的名字代替无意义的数字。双击“dword_407030”来到结构体数据处，在这利用D键、A键或[image]按钮（数组的项数设置为20），将数据重新定义一下。结果如下：

[image]

打开结构体窗口，按Insert键增加一个新的结构体类型，命名为student，如图3.25所示。

[image]

图3.25　创建新的结构体

按D键加入数据（如id、age），重复按D键可在db、dw、dd间切换，直到变成dd，表示是dword类型。而按A键加入的ASCII字符（如name）为结构的成员，此处数组大小填20，如图3.26所示。如果要创建一个大小可变的结构体时，可以将此处自定义的数组元素大小设为0。当增加新结构成员时，IDA会自动加以命名，如field_0等；可按N键

修改新结构成员的名字。新定义的结构体如图3.27所示。

[image]

图3.26　定义结构体中的数组

[image]

图3.27　新定义的结构体

现在将鼠标指针放在407030这一行上，然后执行菜单“Edit/Structs/Struct var”命令，出现如图3.28所示的窗口，以供选择结构体类型。

[image]

图3.28　选择结构体类型

选择student结构体，单击“OK”按钮即可将数据纠正过来。同样方法，重复执行“Struct var”命令，将40704C这行数据转换成student类型。转换后的数据如下：

[image]

最后，可以在操作数类型中重新定义现有的数据。选中需要重新定义的数据，如[esi＋18h]，单击菜单“Edit/Operand types/Offset/Offset(Struct)”或按T键执行结构偏移功能，再选择student结构体。依次将[esi]、[esi＋4]重新定义，最终的效果见图3.29。

[image]

图3.29　用结构体修饰过的代码

如果结构体的成员数较多，不必一个个替换，IDA提供了批处理操作，可以一次操作就能替换全部。方法是选择需要替换的整个代码，执行“Offset(Struct)”菜单命令或按T键，打开结构偏移设置窗口，见图3.30。右边窗口显示与ESI有关系的所有操作。结构各成员前面不同的符号表示经过计算后的状态，“钩”号就表示完全匹配。

[image]

图3.30　设定批量结构偏移

IDA还可以从已经分析好的数据中来建立结构体。在地址407030处，选择一块已重新组织过的数据，用菜单“Edit/Structs/Create struct from data”命令来创建结构体，如图3.31所示。

[image]

图3.31　从现有数据中自动创建结构体

结构体在结构体里又怎样？是的，这是可能的。首先定义结构体，然后在高于这个结构体的另一个级别上，按“Alt + Q”键嵌入另一个实例而成为该结构体的成员。结果如下：

[image]

IDA中可以像定义标准结构体那样来定义共用体（Union），IDA认为共用体是一种特殊的结构体，因此其操作方法与结构体差不多。在结构体（Structures）窗口中，按Insert键打开创建结构体窗口，见图3.25，将选项“Create union”选中，即可创建共用体。共用体的偏移量用菜单“Edit/Structs/Select union member”来操作。

IDA虽然可以使用手工方式建立各类结构，但操作并不方便，从C文件头中导入结构是最好的选择。自己建立的头文件通过积累，将来在遇到类似需要的时候，可以快速地导入到IDA中。方法是：从菜单“Load file/Parse C header file”加载自定义的头文件，然后就可以在结构体窗口使用导入的结构名了。

3.2.13　枚举类型

可以在反汇编时用IDA去动态地定义与操作枚举类型（Enumerated Types）。看看下面这段简单的C程序。

[image]

用IDA反汇编后得到的是一些没有意义的数字，如图3.32所示。

[image]

图3.32　定义枚举前的代码

可以用枚举类型来表示这些数字，执行菜单“View/Open subviews/Enumerations”或单击工具栏上的[image]按钮打开枚举窗口。按Insert键插入一个新的枚举类型，取名“weekday”。在新建的weekday枚举类型中按N键添加枚举成员（见图3.33），0对应MONDAY，1对应TUESDAY，依此类推。

[image]

图3.33　添加枚举成员

最后，可以在操作数类型中重新定义现有的数据。将光标移到需要重新定义的数据处，执行菜单“Edit/Operand types/Enum member”或单击[image]按钮或按M键转换成指定的枚举成员，也可以选中数字后，执行右键菜单“Symbolic constant”命令。处理后的代码如图3.34所示，IDA用MONDAY，TUESDAY等代替了无意义的数字0，1等，使代码非常易读。

[image]

图3.34　直接显示枚举成员

IDA也支持Bit-fields，IDA认为Bit-fields是一种特殊的枚举类型。在图3.35中创建枚举类型时，将选项“Bitfield”勾上，即可创建Bit-fields类型。

[image]

图3.35　创建一个枚举类型

3.2.14　堆栈变量

先看看W32Dasm反汇编ReverseMe的一段代码。很明显，下面的这段代码可以改善，参数的传递并不明显，唯一知道的只是一些数据递交给这函数。

[image]

IDA会自动认出哪些参数放到堆栈中，如下所示：

[image]

与前面一样，在IDA里，堆栈变量也是可以给出有意义的名称的。在任何函数堆栈上（如Msg上）双击鼠标左键或按“Ctrl + K”键打开堆栈窗口，将鼠标移到tagMSG上，将显示其各结构成员。

3.2.15　IDC脚本

IDA能够称之为专业的一个重要因素是IDC，它是一种嵌入式语言，IDC的存在极大地提高了IDA的扩展性，使得IDA中许多重复的任务可以交由IDC来完成，在令其自动化的同时，又可对一些特殊情况进行控制。

IDC本身是一种类C的语言的脚本控制器，语法基本与C类似，简单易学。所有的IDC脚本都有一条包含idc.idc文件的语句，其为IDA的标准库函数，各函数的含义参考IDA帮助文件。变量定义形式为：auto var。其他一些逻辑、循环等语句与C类似。

实例**1**　查看输入函数

现在编写一个查看输入表的IDC脚本程序。通常，某些编译器给输入表所在的区段生成的默认名字是.idata。实际上，输入表可以放在任意一个区段中，.idata不是输入表的标志，定位输入表的正确方法是根据PE文件的结构特征。此例目的是演示脚本编写，就不考虑特殊情况了，假设输入表在.idata块中。

下面这段IDC程序查看PE文件的输入函数：

[image]

单击菜单“File/IDC file”，打开脚本文件选择窗口，选中“Imports.idc”文件，出现如图3.36所示的窗口。单击左边的“Imports”按钮查看Imports.idc文件，单击右边的“Imports”按钮执行IDC脚本文件。如果打开了多个脚本，将依次排列，在相应按钮上单击右键可删除相应脚本。脚本输出结果在IDA提示窗口中。

[image]

图3.36　脚本窗口

实例2　用IDC分析加密代码

一些特殊反汇编任务需要IDC的协助，如对代码段进行加密的程序，可以用IDC先写一段解密的代码，在解密后再反汇编就可以得到正确的反汇编结果。用IDA反汇编光盘映像文件中的encrypted.exe程序。先分析程序入口点处代码：

[image]

分析一下“call 401060”呼叫的子程序，对有疑问的地方IDA以红颜色显示。很明显，下面这段代码毫无意义：

[image]

稍微分析一下，就会发现原来程序是调用“call 401080”子程序对上述代码解密的，解密后才执行。解密代码如下：

[image]

这段代码利用了SMC技术（Self Modifying Code，自己修改自己的代码），就是在可执行文件中保存着加密的数据。只有程序在运行时，才会由程序在某处由一段还原代码来解密这段加密数据。然后，程序才执行这段还原后的代码。具体过程如下：

[image]

具体来看看这段代码执行。首先指令“mov eax,401060”将待解密数据的首地址放进eax寄存器，接着“mov bl,[eax]”将待解密的数据取一位放进bl，然后对该数据做异或操作（xor bl,01），指令“inc eax”指向待解密程序的下一个字节，“cmp eax,00401074”检查待解密的数据是否结束。归纳一下，这段程序的功能就是将401060到401074之间的数据与1做异或运算，从而还原数据。在此利用一段IDC子程序去模拟这段解密过程，还原后的数据就是在IDA中可以“看”到的真实代码。

[image]

单击菜单“File/IDC file”将此段程序载入IDA中，再单击菜单“File/IDC command”或按“Shift + F2”键打开IDC命令执行窗口（见图3.37），

以十六进制形式输入命令“decrypt(0x00401060,0x15,0x1);”。

[image]

图3.37　IDC命令执行窗口

执行完语句后，这组字节就被解码了：

[image]

最后就是手动通知IDA重新分析这段代码。先用U命令将所有的代码以数据形式显示出来，然后在401060处用C命令指示IDA将此段代码重新分析。结果如下：

[image]

遗憾的是，大多数加密程序比一个简单的xor更复杂。解决方法是一样的，只是操作更复杂些。

在IDA中对于使用SMC或其他技术加密代码，也可以使用其他方法将数据解码，如OllyDbg动态调试，然后通过IDA的“Additional binary file”功能将解码文件重新加载，这样的做法比IDC脚本来得更有效。具体使用请参考13.10“静态脱壳”一节。

IDA自带的IDC脚本语言比较简陋，写起脚本来非常不方便。于是一些爱好者写了专门的脚本开发工具来改善，如IDApython、IDAperl等，这些工具功能比较强大，并且有些还支持调试的功能。

3.2.16　FLIRT

IDA的另一项卓越的能力是库文件快速识别与鉴定技术（Fast Library Identification and Recognition Technology，FLIRT）。这项技术使IDA能在一系列编译器的标准库文件里自动找出调用的函数，使反汇编清单清晰明了。

通常的反汇编软件对于各种开发库显得无能为力，只能给出其反汇编结果，而不能给出库函数的名称。例如标准的C函数strlen()，在反汇编中它可能显示为“call 406E40”，这样的反汇编结果虽然是正确的，但却是无意义的。IDA的FLIRT技术可以使反汇编结果中正确标记出所调用的库函数名称，例如以“call strlen”形式显示，这样就极大地提高了反汇编结果的可读性。许多反汇编器都有类似的函数注解功能，但通常限于所调用DLL的输出函数。而IDA试图扩展到包含尽可能多的函数，如流行的开发库MFC，OWL，BCL等。FLAIR的思想是为每个可标识的库函数创建一个“签名”，以使IDA在分析汇编代码时能认知和标记它。

IDA通常可识别一些编译器，但不一定都会成功，如反汇编一些特定版本编译器产生的程序，像微软的记事本程序。另一种情况导致识别失败的原因是程序里编译器的资料被删除了，如用高级语言编写的病毒程序。最后一种情况是编译器的不支持而失败。

如果IDA对程序的编译器支持，但FLIRT没有自动识别出，此时可强制使用其编译器特征文件。在这以一个Delphi 5写的程序演示一下，

Delphi 5的签名文件d5vcl.sig在IDA\SIG目录中。运行IDA打开光盘映像文件中提供的Delphi5.exe程序，反汇编后某段代码如下，注意斜体字体代码处。

[image]

从上面的代码可见，FLIRT没有起自动识别作用，必须强制使用Delphi 5的签名文件d5vcl。单击菜单“View/Open subviews/Signatures”或按“Shift + F5”键或单击[image]按钮打开签名窗口，在该子窗口中单击鼠标右键选择“Apply new signature”或按Insert键打开库文件列表窗口（见图3.38），选中“Delphi 5 Visual Component Library”即可激活Delphi 5的签名文件。

[image]

图3.38　签名列表窗口

[image]

图3.39　应用了Delphi 5签名

应用新的签名文件后，IDA会自动重新分析全部代码，图3.39中的#func栏的数字会跳动，表示已分析了多少个Delphi函数。如果由于某些原因，IDA没自动分析，也可单击菜单“Options/Analysis”打开分析配置选项，单击“重新分析程序（Reanalyse program）”按钮。所有的函数被确认后，反汇编出来的结果就比较有意义了。重新分析后的代码如下：

[image]

IDA为了方便用户自己制作识别库文件，提供了FLIRT数据库生成工具FLAIR，这将给反汇编工作带来更多的便利。该工具是命令行工具，先用pcf.exe将*.lib生成*.pat文件。命令如下：

[image]

再用sigmake.exe文件转换成签名文件。命令如下：

[image]

如果只有dll文件，则反汇编该dll，用IDB2PAT插件生成.pat文件，再用sigmake生成.sig文件。FLAIR更详细用法，请阅读其自带的帮助文档。

3.2.17　插件

IDA支持插件模块，从而可以扩展其功能并自动完成某些任务。一些功能IDC与插件都能完成，但插件提供了更多的函数，还可使用IDA SDK以外的函数，因此可以完成一些复杂的任务。开发插件，需要IDA

软件开发集成工具包（SDK），可以与IDA产品一起获得。

例如Hex-Rays Decompiler插件，它是一款将汇编直接反编译成高级语言的插件，功能十分强大，虽然反编译效果还待改进，但已大大提高了代码的可读性。使用很简单，反汇编好目标程序后，按F5键即可得到源代码。

3.2.18　其他功能

1．图形化模式

从5.0版本开始，IDA又新支持了一种全新的分析模式——图形化模式。这种模式比过去的文本模式有更好的可视性，更容易看清函数的代码流程。

在图形化模式下，当前的函数是由节点和流程连线组成的。用户可以通过空格键切换文本模式和图形化模式。由于在图形化模式下，函数的所有代码可能在屏幕中显示不全，IDA中专门提供了总览窗口（Graph overview），通过单击窗口中感兴趣的位置，可以快速定位该代码片断。用户也可以通过总览窗口了解目前正在分析的局部代码在函数中所处的位置。

2．加载符号文件

IDA 5.0以上支持在菜单“Load file/PDB file”里加载DBG和PDB文件的功能。此项功能对于系统软件而言，非常强大，机器只要联网，IDA会自动到微软的网站去寻找最适合当前文件版本的PDB或DBG文件。

3．**API**帮助关联

将idagui.cfg中的“HELPFILE”指向本地Windows API帮助文件即可实现API帮助关联。这样，用户在分析文档时，看到不熟悉的API时，选中该API，并按“Ctrl + F1”键，即可获得该API的帮助信息。

4．文件的输出

反汇编代码输出功能在菜单“File/Produce file”处。可以按MAP，ASM，LST，EXE，DIF等格式输出。

（1）MAP文件

输出MAP文件，MAP是一个文本文件，记录了各函数等符号信息。

（2）ASM文件

仅输出汇编代码部分，每行代码前无地址。若仅想输出一段代码，选择要保存的代码，然后执行“File/Create ASM file”即可。

（3）Dump database to IDC file

这个命令将当前IDA数据变化记录到IDC文件中，以供恢复当前数据使用。每个IDA新版本都有它自己的数据格式，互不兼容。利用该命令可将低版本中的工作记录转换到高版本中。首先在低版本中利用此命令输出一个IDC脚本文件，然后在IDA新版本中重新装载分析原文件，完成后按F2键打开该IDC脚本执行，结束后，所有的注释、重命名等记录都导入到新版本的IDA中。

（4）Dump typeinfo to IDC file

该命令主要将一些用户自定义数据类型保存到IDC文件中，如关于结构体、枚举等的用户自定义类型。可以利用此命令将一个程序的数据类型导入到另一个程序中。

3.2.19 小结

IDA在提供专业的反汇编能力的同时，也提供了很多相当优秀的辅助功能，如制作流程图和动态调试等。动态调试与静态反汇编本身的紧密联系使得IDA分析程序更得心应手了。

IDA Pro是目前最好的反编译器，改变了人们反汇编的方法，它更像一个智能的反汇编工具。IDA从对程序代码进行反汇编开始，然后分析程序流程、变量和函数调用等。IDA很难使用，并需要有关程序行为的高级知识，但它的技术层次反映了逆向工程真正的本质特征。IDA提供了操纵程序特征的完整的API调用，因此用户能定性分析。对于Windows SDK开发的程序，用IDA配合一般都能逆向出源代码，特别是Hex-Rays Decompiler插件的出现，使得源代码获得更轻而易举。

IDA也不是十全十美的，其最大的缺陷是其反汇编的速度，由于各种高级功能的引进，IDA在反汇编速度一项上，落后其他大多数反汇编工具，其较慢的反汇编速度成为IDA被批评的最主要因素。另一个缺陷来自于用户界面，由于IDA的强大交互性，导致IDA的界面过于专业，大量的窗口、反汇编术语可能导致初级用户无所适从；但反过来也可以说，这样的用户界面又再次加强了IDA在专业领域的地位。

3.3 可执行文件的修改

IDA适合分析文件，若要对文件进行编辑修改，则需要专门的十六进制工具了。常用的有Hiew，HexWorkshop，WinHex等。各工具都有其特色，HexWorkshop提供了文件比较功能；WinHex可以查看内存映像文件；Hiew可以在汇编状态下修改代码。上一章的OllyDbg也可修改保存代码，并且汇编语法更加强大，因此读者可以跳过本节。

本节主要讲解如何利用Hiew修改PE文件中的指令代码，其他工具读者可自行摸索操作。

1．安装

将Hiew压缩包解压到指定目录中就可完成其安装。为了使用方便，建议配置一下hiew.ini文件：

[image]

StartMode项默认是Text。建议将其设置为Code，这样一进入Hiew就自动切入代码界面。Hiew是控制台用户界面，不够友善。运行方式有：

- 将要修改的文件拖到hiew.exe上；
- 将hiew.exe快捷方式放进SendTo目录（推荐）。

由于屏幕属性等原因，Hiew在某些系统上运行不了。此时打开命令提示符窗口，在“属性／布局／屏幕缓冲区大小”中将高度改成25。或新

建一个pif文件，把代码页改成英文，把屏幕高度改成25。

2．修改指令的操作

用Hiew打开实例文件ReverseMe.exe。按回车键，将在3种模式间循环：文本（Text）、十六进制（Hex）和汇编代码（Decode）。按F1键将列出各模式下的帮助清单。

切换到汇编代码模式下，文件以反汇编代码形式显示，用户可以在汇编状态下修改和分析程序，这是Hiew的最强大之处。

Hiew的汇编模式语法如下：

• “byte/word/dword/pword/qword/tbyte”可简写成“b/w/d/p/q/t”；

• 所有的数字都是十六进制，所以“h”操作符是可选的；

• 以A，B，C，E或F开头的十六进制数据，必须加个前缀0，如0A1256等；

• 无条件跳转指令“jmp xxxx”将转换成“0E9 xx xx...”形式，因此近转移jmp（0EB）需要按如下形式输入：jmp short xxxxx（或jmps xxxxx）。远转移的形式是：jmp xxxxx，其中xxxxx是文件偏移地址。在这一点上要格外小心，若将近转移写成长指令形式，或将偏移地址写成虚拟地址，当时在Hiew可能看不出来，但一执行就会出错。（从这点看出OllyDbg的反汇编引擎更强大，自动处理远转移和近转移。）

在此以修改ReverseMe为例，讲解一下Hiew的文件修改功能。

（1）为ReverseMe实例增加水平和垂直滚动条

滚动条显示是由CreateWindowEx函数的样式参数（dwStyle）控制的，具体参数可以查API手册。用IDA反汇编ReverseMe后，在输入表窗口查找CreateWindowExA，双击来到如图3.40所示的代码处。

[image]

图3.40　IDC命令执行窗口

双击WinMain(x,x,x,x)＋AF↑r这里，会来到调用CreateWindowExA代码处：

[image]

要显示滚动条，只需给dwStyle参数再加上两个值：WS_HSCROLL和WS_VSCROLL即可。查VC头文件WINUSER.H得知：WS_HSCROLL定义为00100000h，WS_VSCROLL定义为00200000h。这些参数以逻辑或（OR）做运算（可用Windows自带的计算器）：

[image]

经过上面计算得知，将0040109E一行指令改为“push 00FF0000”，即可加上滚动条。用Hiew打开ReverseMe，切换到汇编代码模式（见图3.41），此时主窗口显示的地址是虚拟地址。按F5键可跳转到指定的地址，此时输入文件的偏移地址：109E，再按回车键就可跳到40109E

处。

[image]

图3.41　Hiew运行界面

[image]技巧：按F5键后，也可输入虚拟地址，格式是在地址前加个点号“.”，如本例输入“.40109E”。

跳到40109E处，按F3键进入编辑状态，此时主窗口显示的地址是文件偏移地址。此时可直接修改机器码的数据，也可在指定行上按回车键或按F2键修改汇编代码（见图3.42）。

[image]

图3.42　进入汇编代码编辑状态

输入push 0FF0000后（记得F前加个0），按回车键跳到下一行，按Esc键返回。确认无误后，按F9键存盘。再运行ReverseMe，出现了水平和垂直滚动条。

（2）数据块加密

此项功能可以对数据或代码进行一些简单的加解密运算。用Hiew打开IDA一节的encrypted程序，在十六进制模式或代码模式下，按F3键进入编辑状态，然后按F7键进入加密模式界面，再按回车键输入指令（见图3.43）。数据的运算可以按byte/word/dword格式进行，按F2键循环切换。

[image]

图3.43　数据加解密操作

指令代码不支持跳转指令，用loop指令代替，其含义是“jmp/stop”。rol/ror指令要求两个操作符大小需相等。32位寄存器不支持“div”、“mul”指令。AL/AX/EAX寄存器中存放的是待运算的数据。下面是几个示例。

- 与01h字节进行异或运算：

[image]

- 除以2：

[image]

- 计算ax =（ax × 3）/ 2：

[image]

输入指令后，按F9键可将当前的算式保存到文件中，下次需要时按F10键调出。然后按F7键（Exit）或按Esc键回到编辑界面。将光标移到

需要修改的数据处，按F7键（Crypt），对当前光标所在的数据进行计算。当然，若是简单的XOR运算，可直接用F8键的XOR运算功能。

3.4 静态分析技术应用实例

学到这里，读者所掌握的理论知识已经可以进行一些简单的应用了，如逆向工程、病毒分析等。

3.4.1 解密初步

现在的软件一般采取人机对话方式，因此从提示信息入手很快就能找到要害。在这里，利用本书光盘映像文件中的一个小程序来讲解。为了叙述方便，将为了练习解密而特别编写的这个程序称为CrackMe。运行CrackMe，随意输入几个字符，出现如图3.44所示的提示信息。

[image]

图3.44　出错提示信息

[image]

图3.45　串式参考窗口

用W32Dasm或IDA对CrackMe.exe进行反汇编，在串式数据参考中找提示信息，找到“序列号不对，重新再试一次！”一句，如图3.45所示。双击它来到相关代码处：

[image]

现在必须从这行起向上找，直到找到cmp，jne，je，test等比较、跳转指令。注意这一行代码：

[image]

004010C9(C)表示指令由4010C9转到此，来到这行代码处：

[image]

注意如下几句，比较关键：

[image]

看明白了吗？要让程序接受任何注册码就只要把jne（不相等就跳）改成je（相等就跳），或改成空指令nop即可。

启动Hiew，打开CrackMe.exe，进入代码模式，按F5键输入虚拟地址“.4010C9”（或输入偏移地址10C9），将指令jne004010E8改成nop指令，由于jne指令是2个字节，因此用2个nop指令替代。具体如下：

[image]

此时输入任何序列号，CrackMe.exe均提示注册成功。这种跳过算法分析，而直接修改关键跳转指令使程序注册成功的方法，常被解密者称为爆破法。

此例的算法只是将用户输入的序列号与参照值比较，以判断真假。其真正的核心就是一句比较指令：

[image]

这种直接比较的程序参照值一般会存在程序中，大多数情况下，编译器会将初始变量放在数据区块（.data区块）中，用十六进制工具打开文件，跳到.data块处，会发现一个数字“9981”，这个就是正确的密码，如图3.46所示。

[image]

图3.46　用十六进制工具查看.data区块

所以，建议软件开发者在注册码验证过程中，不要让正确的注册码直接出现在程序中。另外，也不要采用明显的提示信息，以便被解密者利用，快速找到判断核心。

3.4.2　逆向工程初步

一般将为了练习逆向工程而特别编写的程序称为ReverseMe，本例ReverseMe01有如下要求：

- 移去“Okay, for now, mission failed”对话框；
- 显示一个MessageBox对话框，上面有读者输入的字符；
- 再次显示一个对话框，以告知输入序列号正确还是错误：“Good/Bad serial”；
- 将按钮标题由“Not Reversed”改为“-Reversed-”；
- 使序列号为“pediy”。

1．移去“mission failed”对话框

用IDA打开ReverseMe01进行反汇编，查看Strings窗口，双击字符“Okay, for now, mission failed!”转到定义此字符串的代码处。再双击后面交叉参考转到调用此字符串的代码处：

[image]

按要求，可以用nop指令代替此处的代码，也可用一句跳转指令跳过此提示窗口部分。用Hiew打开ReverseMe01，来到0040123B处，将此处改为“jmp 0040124E”（注意是近转移，输入形式是：jmps 64E）。

2．将输入字符显示到对话框

取得编辑框字符的函数有GetWindowText、GetDlgItemText等，可在程序的输入表中查看。在IDA中，输入、输出等函数显示在Name窗口

中，在Name窗口中双击GetWindowTextA来到调用处，会发现有两处调用了此函数，其中第一处比较可疑：

[image]

从上面分析可知，只要将00401211指令NOP掉（即将机器码改成9090），程序就可执行这段程序。GetWindowTextA函数将文本控件内容取出放进缓冲区4030CC中，MessageBoxA函数从该缓冲区中读取文本并显示到消息框中。

3．修改字符

用HexWorkshop或Hiew查找字符“Not Reversed”，将其改成“-Reversed-”（不要忘记字符串是以00h结尾的）。再将“Good Number”改成“Good Serial”，“Bad Number”改成“Bad Serial”。最后一步，找个空间存放序列号字符：“pediy”，省事的办法是将窗口类名（ClassName）“Supremedickhead”改为“pediy”，其虚拟地址为403000h。

4．完成序列号验证

现在开始写检测序列号的代码。一个较好的地方是在401270h处，该处是原来的算法代码空间。

[image]

到现在，大家基本掌握了静态分析相关工具的使用了，但只是第一步，需要掌握一定的逆向分析技能，才能更好地调试分析程序，剖析软件的最深处。

第4章　逆向分析技术

将可执行程序反汇编，通过分析反汇编代码来理解其代码功能，如各接口的数据结构等，然后用高级语言重新描述这段代码，逆向分析原软件的思路。这个过程被称做“逆向工程（Reverse Engineering）”，或者有时只是简单地称作“逆向（Reversing）”。这是一个很重要的技能，需要扎实的编程功底和汇编知识。逆向分析的首选工具就是IDA，其中它的一款插件Hex-Rays Decompiler能完成许多代码反编译的工作，逆向时可以作为一款辅助工具参考。

逆向工程可以让人们了解程序的结构以及程序的逻辑，因此利用逆向工程可以深入洞察程序的运行过程。一般所谓的“软件破解”只是逆向工程中非常初级的一部分。本节探讨的代码分析技术是基于IA-32处理器体系结构的。

4.1　启动函数

在编写Win32应用程序时，都必须在源码里实现一个WinMain函数。但Windows程序执行并不是从WinMain函数开始的，首先被执行的是启动函数相关代码，这段代码是编译器生成的。启动代码完成初始化进程，再调用WinMain函数。

对于Visual C＋＋程序来说，它调用的是C/C＋＋运行时启动函数，该函数负责对C/C＋＋运行库进行初始化。Visual C＋＋配有C运行库的源代码，可以在crt\src\crt0.c文件中找到启动函数的源代码（安装时Visual C＋＋必须选取安装源代码选项）；而用于控制台程序的启动代码存放在crt\src\wincmdln.c文件中。

所有的C/C＋＋运行时启动函数的作用基本都是相同的：检索指向新进程的命令行指针，检索指向新进程的环境变量指针，全局变量初始化，内存堆栈初始化等。当所有的初始化操作完毕后，启动函数就调用应用程序的进入点函数。调用WinMain函数如下所示：

[image]

当进入点返回时，启动函数便调用C运行库的exit函数，将返回值（nMainRetVal）传递给它，进行一些必要处理，最后调用系统函数ExitProcess退出。

下面是一个Visual C＋＋编译的程序，程序启动代码的汇编代码如下：

[image]

开发人员也可以修改启动源代码，这样会造成即使是同一编译器，而生成的启动代码也会不同。其他一些编译器，如Delphi、BorLand C＋＋开发包中都有相应的启动代码，感兴趣的读者可以自己研究一下。

[image]注意：在分析程序过程中，启动代码可以略过，直接将重点放到WinMain

函数体内。

4.2 函　数

程序都是由不同功能的函数（又称子程序、过程或类似概念的东西）组成的，因此逆向分析中将重点放在函数的识别以及参数的传递上是明智的，这样可以将注意力集中在某一段代码上。函数是一个程序模块，用来实现一个特定的功能。一个函数有如下几部分：函数名、入口参数、返回值、函数功能。

4.2.1 函数的识别

程序中通过调用程序调用函数，而在函数执行完后又返回调用程序继续执行。函数是如何知道要返回的地址呢？实际上，调用函数的代码保存了一个返回地址，并连同参数一起传递给被调用的函数。有多种方法实现这个功能，在绝大多数情况下，编译器都使用CALL与RET指令来调用函数与返回调用位置。

CALL指令与跳转指令功能类似，所不同的是，CALL保存返回信息，即将其之后的指令地址压入堆栈的顶部，当遇到RET时返回到这个地址处。也就是说，CALL指令给出的地址就是被调用函数的起始地址，RET指令结束函数的执行（当然不是所有的RET指令都标识函数的结束）。这一机制使得很容易地把函数调用和其他跳转指令区别开来。

于是可以通过定位CALL机器指令或利用RET指令结束的标志来识别函数。CALL指令的操作数就是所调用函数的首地址。先看一个例子：

[image]

编译结果的大致情况如下：

[image]

这种函数直接调用方式，使得程序很简单，所幸大部分情况都是这样的。也有一些情况，程序调用函数是间接调用的，即通过寄存器传递函数地址或动态计算函数地址。例如：

[image]

4.2.2 函数的参数

函数传递参数有三种方式：堆栈方式、寄存器方式以及通过全局变量进行隐含参数的传递。如果参数是通过堆栈传递的，就需要定义参数在堆栈中的顺序，并约定函数被调用后，由谁来平衡堆栈。如果参数是通过寄存器传递的，就要确定参数存放在哪个寄存器中。每种机制都有其优缺点，并且与使用的编译语言有关。

1．利用堆栈传递参数

堆栈是一种“后进先出”的存储区，栈顶指针ESP指向堆栈中第一个可用的数据项。调用函数时，调用者依次把参数压栈，然后调用函数，

函数被调用以后，在堆栈中取得数据，并进行计算。函数计算结束以后，或者调用者，或者函数本身修改堆栈，使堆栈恢复原样（即平衡堆栈）。

在参数传递中，有两个很重要的问题必须得到明确说明：当参数个数多于一个时，按照什么顺序把参数压入堆栈？函数结束后，由谁来平衡堆栈？这些都必须有个约定，这种在程序设计语言中为了实现函数调用而建立的协议称为调用约定（Calling Convention）。这种协议规定了函数中的参数传送方式、参数是否可变和由谁来处理堆栈等问题。不同的语言定义了不同的调用约定，常用的调用约定见表4-1。

表4-1　调用约定

[image]

注：VARARG表示参数的个数可以是不确定的；stdcall如果使用VARARG参数类型，则是调用程序平衡堆栈，否则是被调用程序平衡堆栈。

C规范（即__cdecl）函数参数按照从右到左的顺序入栈，由调用者负责清除堆栈。__cdecl是C和C＋＋程序的默认调用约定。C/C＋＋和MFC程序默认使用的调用约定是__cdecl，也可以在函数声明时加上__cdecl关键字来手工指定。

PASCAL规范按从左到右的顺序压参数入栈，要求被调用函数负责清除堆栈。

stdcall调用约定是Win32 API函数采用的约定方式，它是“标准调用（Standard CALL）”之意，它采用C调用约定的入栈顺序和PASCAL调用约定的调整栈指针方式，即函数入口参数按从右到左的顺序入栈，并由被调用的函数在返回前清理传送参数的内存栈，函数参数个数固定。由于函数体本身知道传进来的参数个数，因此被调用的函数可以在返回前用一条“ret n”指令直接清理传递参数的堆栈。在Win32 API中，也有一些函数是__cdecl调用的，如wsprintf。

为了了解不同类型约定的处理方式，来看这个例子。假设调用函数test1(Par1,Par2,Par3)，按__cdecl、PASCAL和__stdcall的调用约定，其汇编代码如表4-2所示。

表4-2　汇编代码

[image]

可以清楚地看到，__cdecl类型和__stdcall类型是先把右边参数压入堆栈，而PASCAL则相反。在堆栈平衡上，__cdecl类型是调用者用“add esp,0c”指令把12个字节参数空间清除，而PASCAL和__stdcall类型则是子程序负责清除。

函数对参数的存取和局部变量都是通过堆栈来定义的，非优化编译器用一个专门的寄存器（通常是ebp）对参数进行寻址。C、C＋＋、Pascal等高级语言的函数（子程序）执行过程基本都是一致的。情况如下：

（1）调用者将函数（子程序）执行完毕时应返回的地址、参数压入堆栈；

（2）子程序使用“ebp指针＋偏移量”对堆栈中的参数寻址，并取出，完成操作；

（3）子程序使用ret或retf指令返回。此时，CPU将eip置为堆栈中保存的地址，并继续予以执行。

堆栈在整个过程中发挥着非常重要的作用。堆栈是一个“先进后出”的区域，堆栈只有一个出口，即当前栈顶，堆栈操作的对象只能是双字操作数（占4个字节）。例如，按__stdcall约定调用函数test2(Par1,Par2)（有2个参数），其汇编代码大致情况如下：

[image]

因为esp是堆栈指针，所以一般使用ebp来存取堆栈。其堆栈建立情况如下：

（1）此例函数中有两个参数，假设执行函数前堆栈指针的esp为K；

（2）根据__stdcall调用约定，先将参数Par2压进栈，此时esp为K-04h；

（3）再将参数Par1压进栈，此时esp为K-08h；

（4）参数进栈结束后，程序开始执行call指令，call指令把返回地址压入堆栈，这时候esp为K-0Ch；

（5）这时已经在子程序中了，可以开始使用EBP来存取参数了，但为了在返回时恢复ebp的值，用“push ebp”保存ebp的值，这时esp为K-10h；

（6）再执行一句“mov ebp,esp”，ebp被用来在堆栈中寻找调用者压入的参数，这时候[ebp＋8]就是参数1，[ebp＋c]就是参数2；

（7）“sub esp,8”，在堆栈中定义局部变量，局部变量1和2对应的地址分别是[ebp-4]和[ebp-8]。函数结束时，调用“add esp,8”释放局部变量占用的堆栈。局部变量的范围从它的定义到它定义所在的代码块的结束为止，也就是说，当函数调用结束后局部变量便消失。

（8）最后调用“ret 8”指令来平衡堆栈，ret指令后面加一个操作数表示在ret后把堆栈指针esp加上操作数，完成同样的功能。

处理完毕后，就可以开始用ebp存取参数和局部变量了，图4.1说明了这个过程。

[image]

图4.1　函数堆栈创建图

此外，还有一组指令，即enter和leave，它们可以帮助进行堆栈的维护。enter语句的作用就是“push ebp/mov ebp,esp/sub esp,xxx”，而leave则完成“add esp,xxx/pop ebp”的功能。所以，上面的程序可以改成：

[image]

在许多情况下，编译器会按优化方式编译程序，堆栈寻址稍有不同。这时编译器为了把ebp寄存器省下来或尽可能减少代码以提高速度，会直接通过esp对参数进行寻址。esp的值在函数执行期间要发生变化，该变化出现在每次有数据进出堆栈时。要确定是对哪个变量进行寻址，就需要知道程序当前位置的esp值是多少，为此必须从函数的开始部分跟踪。

同样，上例中的test2(Par1,Par2)函数，在VC 6.0里，优化选项设置为“Maximize Speed”，重新编译，其汇编代码可能如下：

[image]

这时程序就用esp来传递参数了，其堆栈建立情况如图4.2所示。

[image]

图4.2　函数堆栈创建图

（1）假设执行函数前堆栈指针esp的值为K；

（2）根据stdcall调用约定，先将参数Par2压进栈，此时esp为K-04h；

（3）再将Par1压进栈，此时esp为K-08h；

（4）参数进栈结束后，程序开始执行call指令，call指令把返回地址压入堆栈，这时候esp为K-0Ch；

（5）这时已经在子程序中了，可以开始使用esp来存取参数了。

2．利用寄存器传递参数

寄存器传递参数的方式并没有一个标准，所有与平台相关的方法都是由编译器开发人员制定的。尽管没有统一的标准，但绝大多数编译器提供商都在不对兼容性声明的情况下，遵循相应的规范，即__fastcall规范。__fastcall，顾名思义，特点就是快，因为它是靠寄存器来传递参数的。

不同编译器实现的__fastcall稍有不同，如Microsoft Visual C＋＋编译器采用__fastcall规范传递参数时，最左边的两个不大于4个字节（DWORD）的参数分别放在ecx和edx寄存器。当寄存器用完后，就要使用堆栈，其余参数仍然按从右到左的顺序压入堆栈，被调用的函数在返回前清理传送参数的堆栈。浮点值、远指针和__int64类型总是通过堆栈来传递的。而Borland Delphi/C＋＋编译器采用__fastcall规范传递参数时，最左边的三个不大于4个字节（DWORD）的参数分别放在eax、edx和ecx寄存器，寄存器用完后，多余参数按照从左至右的PASCAL方式来压栈。另外一款编译器Watcom C总是通过寄存器来传递参数的，其严格为每一个参数分配一个寄存器，默认时第一个参数用eax，第二个参数用edx，第三个参数用ebx，第四个参数用ecx。如寄存器用完，就会用堆栈来传递参数。Watcom C可以由程序员指定任意一个寄存器传递参数，因此，其参数实际上可能通过任何寄存器进行传递。

来看一个Microsoft Visual C＋＋ 6.0编译的__fastcall调用实例：

[image]

用Visual C＋＋编译，Optimizations选项为Default，编译后查看其反汇编代码：

[image]

另一个调用规范thiscall也用到了寄存器传递参数。thiscall是C＋＋中的非静态类成员函数的默认调用约定，对象的每个函数隐含接收this参数。采用thiscall约定时，函数参数按照从右到左的顺序入栈，被调用的函数在返回前清理传送参数的栈，只是另外通过ecx寄存器传送一个额外的参数：this指针。

定义一个类，并在类中定义一个成员函数：

[image]

用Visual C＋＋编译，Optimizations选项为Default，编译后查看其反汇编代码：

[image]

3．名称修饰约定

在C＋＋中，为了允许操作符重载和函数重载，C＋＋编译器往往按照某种规则改写每一个入口点的符号名，以便允许同一个名字（具有不同的参数类型或者是不同的作用域）有多个用法，而不会打破现有的基于C的链接器。这项技术通常被称为名称改编（Name Mangling）或者名称修饰（Name Decoration）。许多C＋＋编译器厂商选择了自己的名称修饰方案。

在VC＋＋中，函数修饰名由编译类型（C或C＋＋）、函数名、类名、调用约定、返回类型、参数等多种因素共同决定。关于名称修饰东西很多，下面仅简单谈一下常见的C编译、C＋＋编译函数名修饰。

C编译时函数名修饰约定规则：

• __stdcall调用约定在输出函数名前加上一个下画线前缀，后面加上一个“@”符号和其参数的字节数，格式为：_functionname@number。

• __cdecl调用约定仅在输出函数名前加上一个下画线前缀，格式为：_functionname。

• __fastcall调用约定在输出函数名前加上一个“@”符号，后面也是一个“@”符号和其参数的字节数，格式为：@functionname@number。

它们均不改变输出函数名中的字符大小写，这和PASCAL调用约定不同，PASCAL约定输出的函数名无任何修饰且全部大写。

C＋＋编译时函数名修饰约定规则：

• __stdcall调用约定以“？”标识函数名的开始，后跟函数名；函数名后面以“@@YG”标识参数表的开始，后跟参数表；参数表的第一项为该函数的返回值类型，其后依次为参数的数据类型，指针标识在其所指数

据类型前；参数表后以“@Z”标识整个名字的结束，如果该函数无参数，则以“Z”标识结束。其格式为：“?functionname@@YG*****@Z”或“?functionname@@YG*XZ”。

• __cdecl调用约定规则同上面的_stdcall调用约定，只是参数表的开始标识由上面的“@@YG”变为“@@YA”。

• __fastcall调用约定规则同上面的_stdcall调用约定，只是参数表的开始标识由上面的“@@YG”变为“@@YI”。

4.2.3 函数的返回值

函数被调用执行完后将向调用者返回一个或多个执行结果，称为函数返回值。返回值最常见的方式是用return操作符，其他的还有通过参数按传引用方式返回值，通过全局变量返回值等。

1．用return操作符返回值

一般情况下，函数的返回值放在eax寄存器中返回，如果处理结果超过了eax寄存器容量，其高32位就会放到edx寄存器中。例如下面这段C程序：

[image]

这是一个普通的函数，它将两个整数相加。这个函数有两个参数，并用了一个局部变量临时保存结果。其汇编实现代码如下：

[image]

2．通过参数按传引用方式返回值

给函数传递参数的方式有两种，即传值和传引用。传值调用时是建立参数的一份拷贝并把它传给调用函数，在调用函数中修改参数值的拷贝不影响原始的变量值。传引用调用允许调用函数修改原始变量的值。在调用某个函数时，当把变量的地址传递给函数时，可以在函数中用间接引用运算符修改调用函数中内存单元中的该变量的值。例如下面这个例子，在调用函数max时，需要用两个地址（或两个指向整数的指针）作为参数，函数会将结果较大的数放到参数a所在的内存单元地址中返回。

[image]

其可能的汇编代码如下：

[image]

[image]

4.3 数据结构

数据结构是计算机存储、组织数据的方式。逆向分析时，确定了数

据结构后，算法就容易得到了。有些时候事情也会反过来，根据特定算法来判断数据结构。本节讨论常见的数据结构以及它们在汇编语言中的实现方式。

4.3.1　局部变量

局部变量是一个函数内部定义的变量，只有在函数内才能使用，如计数器，临时变量等。使用局部变量带来的好处，使程序模块化封装变得可能。从汇编角度来看，局部变量就是在堆栈中进行分配，函数执行完毕后释放这些堆栈，或者直接把局部变量放在寄存器中。

1．利用堆栈存放局部变量

局部变量在堆栈中具体形成过程请参看图4.1，程序用“sub esp,8”语句为局部变量分配空间，用[ebp-xxxx]寻址调用这些变量，而参数调用相对于ebp偏移量是正的，即[ebp＋xxxx]，因此在逆向时比较容易区分开。编译器在优化模式时，则通过esp寄存器直接对局部变量与参数寻址了。当函数退出时，用“add esp,8”指令平衡堆栈，以释放局部变量所占据的内存。有些编译器，如Delphi通过给esp加一个负值来进行内存分配。另外，编译器可能会用“push reg”指令来取代“sub esp,4”指令，以节省几个字节。局部变量分配堆栈几种形式见表4-3。

表**4-3**　局部变量分配与清除堆栈几种形式

[image]

来看下面这个实例是如何用“push reg”指令来取代“sub esp,4”指令的。

[image]

用Microsoft Visual C＋＋ 6.0编译，不优化，其汇编代码如下：

[image]

在函数add()里不存在“sub esp,n”之类指令，程序是通过一句“push ecx”指令来开辟一块堆栈的，然后用[ebp-04]来访问这个局部变量。

局部变量的起始值是随机的，是其他函数执行后留在堆栈中的垃圾数据，因此需要对其初始化。初始化局部变量有两种方法：一种是通过mov指令为变量赋值，如“mov [ebp-04],5”；另一种是使用push指令直接将值压入堆栈，如“push 5”。

2．利用寄存器存放局部变量

除了堆栈占用了2个寄存器外，编译器会利用剩下的6个通用寄存器尽可能有效地存放局部变量，这样可以产生最小的代码，提高程序效率。如果寄存器不够用，编译就会将变量放到堆栈中。逆向分析时，要注意局部变量的生存周期比较短，必须及时确定当前寄存器的变量是哪个变量。

4.3.2 全局变量

全局变量作用于整个程序，一直存在，它放在全局变量的内存区；而局部变量则是存在于函数的堆栈区，当函数调用结束后便消失。在大多数程序中，常数一般放在全局变量中，如一些注册版标记、测试版标记等。

在大多数情况下，在汇编代码中识别全局变量比其他结构要容易得多。全局变量通常位于数据区块（.data）的一个固定地址上，当程序需要访问全局变量时，一般会用一个固定的硬编码的地址直接对内存寻址。一个样例如下：

[image]

全局变量可以被同一文件中所有的函数修改，某个函数改变了全局变量的值，就能影响到其他函数，相当于各个函数间的传递通道。因此就可以利用全局变量传递参数、传递函数返回值等。全局变量在程序的全部执行过程中都占用内存单元，而不像局部变量需要时开辟单元。

来看一个利用全局变量传递参数的实例，具体代码如下：

[image]

用Microsoft Visual C＋＋ 6.0编译，不优化，其汇编代码如下：

[image]

用LordPE打开编译后的程序，查看区块，区块信息如图4.3所示。会发现全局变量4084C0h在.data区块，该区块的属性可读写。

[image]

图4.3　区块信息

这种对内存直接寻址的硬编码方式，比较容易识别出这是一个全局变量。一般编译器会将全局变量放到可读写的区块里，如果放到只读区块里，那么这是一个常量。另外，与全局变量类似的是静态变量，都可以按直接方式寻址等。所不同的是，静态变量作用范围是有限的，仅在定义这些变量的函数内有效。

4.3.3 数组

数组是相同数据类型的元素的集合，它们在内存中按照顺序连续存放在一起。汇编状态下访问数组一般是基址加变址寻址来实现的。

请看下面这个数组访问的实例：

[image]

用Microsoft Visual C＋＋ 6.0编译，优化选项设置为Maximize Speed，其汇编代码如下：

[image]

数组在内存中可以存在于堆栈、数据段以及动态内存中。本例中的a[]数组就保存在数据段.data中，其寻址用“基址＋偏移量”来实现。

[image]

这种间接寻址一般出现在给一些数组或结构赋值情况下，其寻址形式一般是[基址＋n]，其中基址可以是常量，也可以是寄存器，为定值。随着n值的不同，就可对结构中相应单元赋值了。

b[]数组放在堆栈中，这些堆栈是编译时刻进行分配的。数组在声明时可以直接计算偏移地址，针对数组成员寻址是采用实际的偏移量完成的。

4.4　虚函数

C＋＋是一门支持OO的语言，对面向对象的软件开发提供了丰富的语言支持。但要高效、正确地使用C＋＋中的继承、多态等语言特性，必须对这些特性的底层实现有一定的了解。

其实就核心概念而言，C＋＋的对象模型的核心概念并不多，最核心的是虚函数。虚函数是在程序运行时刻定义的函数，虚函数的地址是不能在编译时刻确定的，它只能在调用即将进行之前加以确定。对所有虚函数引用通常都放在一个专用数组——虚函数表（Virtual Table，VTBL）中，每个至少使用一个虚函数的对象里面都具有的虚函数表指针（Virtual Table Pointer，VPTR）。虚函数通常通过指向虚函数表的指针间接地加以调用。

将实例thiscall.exe的普通成员函数改变为虚函数调用，来看看VC在虚函数上面是如何处理的。

[image]

用Microsoft Visual C＋＋ 6.0编译，编译时设置优化选项为Maximize Speed。其汇编代码如下：

[image]

这段代码首先调用new函数分配class所需的内存（new函数的确定是用IDA来识别的），调用成功后，eax保存了分配的内存的指针，然后将对象实例指向CSum类虚函数表（VTBL）4050A0h。查看4050A0h数据，如图4.4所示。

[image]

图4.4　查看VTBL

VTBL里有两组数据：

[image]

进一步看看这两个指针是什么内容，401040h内容如下：

[image]

401050h内容如下：

[image]

原来虚函数是通过指向虚函数表的指针间接地加以调用的，程序仍然用ecx作为this指针的载体传递给虚成员函数，并且利用两次间接寻址，得到虚函数的正确地址，从而执行。

[image]

整个过程如图4.5所示。VPTR是一个虚函数表指针，所有的虚函数的入口点被排成一个表，就是虚函数表（VTBL）。

[image]

图4.5　虚函数的调用实现

4.5　控制语句

在高级语言中用IF-THEN-ELSE、SWITCH-CASE等语句来构成程序的判断流程，不仅条理清楚，并且维护性好。而其汇编代码则复杂得多，会看到cmp等指令的后面跟着各类跳转指令jz，jnz等。识别关键跳转是软件解密的一个重要技能，许多软件用一个或多个跳转来实现注册或非注册功能。

4.5.1　IF-THEN-ELSE语句

将语句IF-THEN-ELSE编译成汇编代码后，整数用cmp指令比较，而浮点值用fcom、fcomp等比较。

语句IF-THEN-ELSE编译后，其汇编代码形式一般如下：

[image]

cmp指令不修改操作数，根据两个操作数的相减结果，影响处理的几个标志，如零标志、进位标志、符号标志和溢出标志。jz等指令就是条件跳转指令，根据a、b的值大小决定跳转方向，更多条件指令见表4-4。

实际上，许多情况下编译器都用test或or之类的较短的逻辑指令来替换cmp指令。一般形式为“test eax,eax”，如eax为0，则其逻辑与运算结果为零，就设置ZF为1；否则设为0。

来看一个实例，源码如下：

[image]

用Microsoft Visual C＋＋ 6.0编译，优化选项设置为Maximize

Speed，其汇编代码如下：

[image]

4.5.2 SWITCH-CASE语句

SWITCH语句是多分支选择语句。SWITCH语句编译后，实质就是多个IF-THEN语句嵌套组合。编译器会将SWITCH编译成一组不同关系运算组成的语句。具体来看一例子：

[image]

用Microsoft Visual C++ 6.0编译，没有优化，其反汇编代码如下：

[image]

[image]

如果编译时设置优化选项为Maximize Speed，其汇编代码如下：

[image]

编译器优化时用“dec eax”指令代替cmp指令，这样指令更短，并且执行速度更快。并且优化后，编译会合理排列switch后各case节点，以最优化方式找到所需要的节点。

如果各case取值表示一个算术级数，那么编译器会利用一个跳转表（Jump Table）来实现。例如：

[image]

编译器编译后，“jmp dword ptr[4*eax+004010B0]”指令就相当于switch(a)，其根据eax的值进行索引，计算出指向相应case处理代码的指针。汇编代码如下：

[image]

在实际程序中，case路径中还可能包含其他跳转分支语句、循环语句等，这样会使问题复杂化。

4.5.3 转移指令机器码的计算

在软件分析过程中，经常需要计算转移指令机器码或修改指定的代码。虽然有许多工具可以做这些事，但掌握其原理和技巧是很有必要的。

根据转移的距离，转移指令有以下类型。

• 短转移（Short Jump）：无条件转移和条件转移的机器码都是两个字节。转移范围是-128～+127字节。

• 长转移（Long Jump）：无条件转移的机器码是5个字节，条件转

移的机器码是6个字节。这是因为条件转移要用2个字节表示其转移类型（如je、jg和jns），其他4个字节表示转移偏移量。无条件转移仅用一个字节就可表示其转移类型（jmp），其他4个字节表示转移偏移量。

• 子程序调用指令（call）：call指令调用有两类。一类是平常经常接触到的，类似于长转移（Long Jump）；另一类其调用的参数涉及到寄存器、堆栈等值，比较复杂，如“call dword ptr[eax+2]”。

条件转移指令的转移范围是16位模式遗留下的，当时为了使代码紧凑些，CPU开发人员只给目的地址分配了一个字节，这样限制了跳转的长度只能在255个字节的范围里。

表4-4列出了常用的转移指令机器码，通过该表就可根据转移偏移量计算出转移指令的机器码。

表**4-4**　转移指令的条件与机器码

[image]

有两个因素制约转移指令机器码：一个是表4-4中列出的转移类型；另一个是转移的位移量。

（1）短转移指令机器码计算实例

例如，代码段中有一条如下所示的无条件转移指令：

[image]

无条件短转移的机器码形式为EBxx，其中EB00~EB7F是向后转移，EB80~EBFF是向前转移。图4.6表示了该转移指令的机器语言，以及用位移量来表示转向地址的方法。由图4.6可见，位移量为3h，CPU执行完“jmp 401005”指令后的EIP值为401002h，然后执行：“（EIP）←（EIP）+位移量”，执行后就跳转到401005地址处。即“jmp 401005”指令机器码形式是“EB 03”（见图4.7）。

[image]

图4.6　短转移举例

[image]

图4.7　机器码组合形式

也就是说，转移指令的机器码形式是：

位移量=目的地址-起始地址-跳转指令本身的长度

转移指令机器码=“转移类别机器码”+“位移量”

（2）长转移指令机器码计算实例

例如，代码段中有一条如下所示的无条件转移指令：

[image]

无条件长转移指令的长度是5个字节，机器码是E9。根据上面公

式，此例转移的位移量为：

00402398h - 00401000h - 5h = 00001393h

由图4.8可见，00001393h在内存中以双字存储（32位）。存放时，低位字节存入低地址，高位字节存入高地址，也就是说，00 00 13 93以相反的次序存入，形成了93 13 00 00存储形式。

[image]

图4.8　长转移举例

[image]

上面两个实例演示转移指令向后转移（由低地址到高地址）。如果是向前转移（由高地址到低地址），计算方法一样。

例如，代码段中有一条如下所示的无条件转移指令向前转移：

[image]

位移量 = 40100h - 402398h - 5h = FFFFEC63h（取后32位）

转移机器码 = “E9” + “63EC FF FF” = E9 63EC FF FF

4.5.4　条件设置指令（SETcc）

条件设置指令的形式是：SETcc r/m8，其中r/m8表示8位寄存器或单字节内存单元。

条件设置指令根据处理器定义的16种条件cc，测试一些标志位，然后把结果记录到目标操作数中。当条件满足时，目标操作数会被置1，否则置0。这16种条件与条件转移指令Jcc中的条件是一样的，如表4-5所示。

表**4-5**　条件设置指令

[image]

条件设置指令可以用来消除程序中的转移指令。在C语言里，常会见到执行以下功能的语句：

[image]

如果允许出现条件分支，编译器会产生如下的代码或者是非常类似的代码：

[image]

如果使用条件设置指令，编译器将会产生不包含条件分支的逻辑判断代码：

[image]

也可用条件传输指令cmov或fcmov去除程序中的转移指令，但是它

们仅被Pentium Pro以后的处理器支持。实现同样功能的代码如下：

[image]

4.5.5　纯算法实现逻辑判断

一些编译器优化时，在不改变原逻辑的情况下，使用数学技巧把源代码中一些逻辑分支语句转换成算术操作，消除或减少程序中出现的条件转移指令，提高CPU的流水线的性能。

来看这段C程序：

[image]

用Microsoft Visual C++ 6.0编译，设置优化选项为Maximize Speed。其反汇编代码如下：

[image]

编译生成的代码没有一句条件转移指令，却实现原程序的逻辑。代码首先用neg指令检验eax是否为0，结果存放在CF标志位中。sbb指令将目的操作数减去源操作数，再减去借位CF（进位），结果送到目的操作数。“sbb eax,eax”这句的结果由CF决定，当CF为1时，eax为-1，否则为0。用伪码来表示：

[image]

接下来两句指令根据eax的值：FFFFFFFFh和0来决定最终结果。当eax是FFFFFFFFh时，计算结果是1；当eax是0时，计算结果为5。

[image]

这类代码比较常见，当知道是条件转移指令优化生成的，还原就比较容易了。

4.6　循环语句

循环是高级语言中可以进行反向引用的一种语言形式，其他类型的分支语句（如IF-THEN-ELSE等）都是由低向高端地址区域走的。因此，通过这点可以较方便地将循环语句识别出来。

如果确定某段代码是循环，就可分析其计数器，一般是用ecx寄存器做计数器，也有用其他方法来控制循环的，如“test eax,eax”等。下面是一段最简单的循环代码：

[image]

上面的汇编代码用高级语言C来描述有以下2种形式：

[image]

再来看一段较复杂的循环，比如这段C程序：

[image]

用Microsoft Visual C++ 6.0编译，没有优化，其反汇编代码如下：

[image]

如果编译时设置优化选项为Maximize Speed，然后看看汇编代码是如何变化的。其汇编代码如下：

[image]

4.7 数学运算符

高级语言中的运算符范围很广，这里只介绍整数的加、减、乘、除运算。编译器如果没优化，这些运算符很好理解，可以参考相关的汇编书籍。本节主要认识一下经编译器优化过的运算符。

4.7.1 整数的加法和减法

一般情况下，整数的加法和减法编译成add和sub指令。编译优化时，比较喜欢用lea指令来代替add和sub指令。lea指令允许用户在一个时钟内完成c = a + b + 78h计算，其中a、b与c都是在寄存器的情况下才有效，会编译成“lea c,[a+b+78]”指令。

加法实例：

[image]

用Microsoft Visual C++ 6.0编译，设置优化选项为Maximize Speed。其反汇编代码如下：

[image]

在这句代码中，lea指令只是一条纯算术指令，它的实际意义等价于“edx = ecx + eax + 78”。

4.7.2 整数的乘法

乘法运算符一般编译成mul、imul指令。这些指令运行的速度比较慢，编译器会尽可能地提高代码的效率，从而倾向于使用其他指令来完成同样的计算。如果一个数是2的幂，那么会用左移指令shl来实现乘法。另外，加法对于提高3，5，6，7，9等数的乘法运算很有用，如：eax*5可以写成“lea eax,[eax+4*eax]”。（lea指令可以实现寄存器乘以2、4或8的运算。）

[image]

用Microsoft Visual C++ 6.0编译，设置优化选项为Maximize Speed。其反汇编代码如下：

[image]

4.7.3 整数的除法

除法运算符一般编译成div、idiv指令。除法运算的代价是相当高的，大概比乘法多消耗10倍的CPU时钟。

如果被除数是一个未知数，编译器会使用div指令，程序执行效率会有所下降。

对于除数/被除数有一个是常量的情况，就会复杂很多。编译器将会使用一些技巧更有效地实现除法。如果除数是2的幂，那么可以用较快的移位指令“shr a,n”来替换，移位指令只需花费一个时钟，其中a是被除数，n是基数2的指数。shr适合无符号数计算，若是符号数则用sar指令。也会根据一定算法用乘法运算来取代除法运算。

具体来看除法的实例：

[image]

用Microsoft Visual C++ 6.0编译，不优化。其反汇编代码如下：

[image]

[image]

除法指令需要用到符号扩展指令cdq，其作用是把eax寄存器中的数视为有符号的数，将其符号位（即eax的最高位）扩展到edx寄存器，即若eax的最高位是1，则执行后edx的每个位都是1，结果edx=FFFFFFFF；若eax的最高位是0，则执行后edx的每个位都是0，结果edx=00000000。这样就把eax中的32位带符号的数变成了edx:eax中的64位带符号的数，以满足64位运算指令的需要，但转换后的值没变。

编译器优化时，会采用乘法运算代替除法运算，这样能提高数倍的效率。不过，对于逆向分析来说，这样的代码比较难理解。

用于优化的公式比较多，最常用的就是倒数相乘。相关公式如下：

[image]

用Microsoft Visual C++ 6.0编译，设置优化选项为Maximize Speed。其反汇编代码如下：

[image]

这段代码就是一个简单的除法运算，编译器优化后的代码比一个idiv指令长，但其运行速度提高了3倍。还有更多的除法优化算法，不同编译器实行的方法也有所不同。

4.8 文本字符串

字符的识别和分析是软件逆向的一个重要步骤，特别是在一些序列号分析过程中，经常遇到各类字符操作。

4.8.1 字符串存储格式

程序中，一般将字符串作为字符数组来处理，但不同的编程语言，其字符存储格式是不同的。常见的字符串类型有C字符串、Pascal字符串等。

1．C字符串

C字符串也称为ASCIIZ字符串，广泛应用于Windows与UNIX操作系统中，Z表示其以“\0”为结束标志。“\0”代表ASCII码为0的字符，如图4.9所示。ASCII码为0的字符不是一个可以显示的字符，而是一个“空操作符”。

[image]

图4.9　C字符串

2．DOS字符串

在DOS下，输出行的函数以“$”字符作为终止字符，如图4.10所示。由于DOS的淘汰，目前已很少见这类字符格式了。

[image]

图4.10　DOS字符串

3．Pascal字符串

Pascal字符串没有终止符，但在字符串的头部定义了一个字节，指示当前字符串的长度。由于只用一个字节来表示字符串的长度，所以字符串不能超过255个字符，如图4.11所示。字符串中的每个字符都属于ANSIChar类型（标准字符类型）。这种类型字符只存在于Borland公司的Turbo Pascal和16位Delphi中。

[image]

图4.11　Pascal字符串

4．Delphi字符串

为克服传统Pascal字符串的局限性，32位Delphi增加了对长字符串的支持。

• 双字节Delphi字符串：表示长度的字段扩展为2个字节，可使字符串的最大长度达到65535，如图4.12所示。

[image]

图4.12　Delphi字符串

• 四字节Delphi字符串：表示长度的字段扩展为4个字节，使得字符串长度可达4GB。目前这种字符类型很少使用。

4.8.2　字符寻址指令

80x86支持寄存器直接寻址与寄存器间接寻址等模式。与字符指针处理相关的有mov、lea等指令。mov指令将当前指令所在内存复制一份到目的寄存器中，其操作数可以是常量，也可以是指针。例如：

[image]

lea意思是“装入有效地址（Load Effective Address）”，它的操作数就是地址，所以“lea eax,[addr]”就是将表达式addr的值放入eax寄存器中。

[image]

lea指令右边的操作数表示一个近指针，指令“lea eax,[401000h]”与“mov eax,401000h”是等价的。

在计算索引与常量的和时，编译器一般将指针放在第一个位置，而不管它们在程序中的顺序。例如这段初始化代码：

[image]

编译器不仅广泛地用lea指令来传递指针，而且常用来计算常量的和，其等价于add指令。也就是说，“lea eax,[eax＋8]”等价于“add eax,8”，不过lea指令的效率将远远高过add，这种技巧可以使多个变量的求和在一个指令周期内完成，同时可以通过任何寄存器返回结果。

4.8.3　字母大小写转换

大写字母的ASCII码范围是41h～5Ah，小写字母的ASCII码范围是61h～7Ah，它们之间的转换就是ASCII码的值加减20h。

下面这段汇编代码的功能是将小写字母转换成大写字母：

[image]

这段代码先用“a”来比较，如小于“a”，可能是大写字母或其他字符；然后再与“z”比较，如果大于“z”，则不是小写字母，并且不处理。如果确定是小写字母，将该字符的ASCII码减20h，即可转换成大写字母。

另外，还有一种大、小写字母转换的方法。图4.13显示的是大写字母“A”与小写字母“a”的二进制形式，如果第5位是0，则是大写字母；如果是1，则是小写字母。

[image]

图4.13　字母的二进制形式

因此，下面代码也能实现大、小写字母的转换：

[image]

4.8.4　计算字符串的长度

高级语言里会有特定函数来计算字符串的长度，如C语言中常用strlen()函数计算字符串的长度。strlen()在优化编译模式下的汇编代码如下：

[image]

这段代码使用串扫描指令scasb操作，把AL的内容与EDI指向的附加段中的字节逐个比较，最后把EDI指向的字符串长度保存在ecx中。

4.9　指令修改技巧

在软件分析过程中，为了优化原程序或在一定空间里增添代码，需要一定的指令修改技巧。表4-6列出了常用指令修改技巧，在以后的实践操作中经常用到它们。

表**4-6**　常用指令修改技巧

[image]

[image]

很多指令针对eax被做了优化，要尽可能多地使用eax。例如，“xchg eax, ecx”只需要1个字节，而用其他寄存器则需要2个字节。

第3篇　解密篇

■ 第5章　常见的演示版保护技术
■ 第6章　加密算法

一些软件开发人员对软件保护方案的策划与实施很不以为然，他们往往自以为是的保护在解密者眼中不堪一击。希望本部分能让这些开发人员了解一些软件攻击的方法，以便更好地保护自己的作品。

第5章　常见的演示版保护技术

本章讲述一些常用的软件保护技术，对其优缺点进行分析，并给出软件保护的一般性建议。软件开发者将会从中获得一些有益的启发，以便更好地保护自己的智力成果。

5.1　序列号保护方式

先来看一看在网络上大行其道的序列号（又称为注册码，本书不再对此进行区分）保护的工作原理。当用户从网络上下载某个共享软件（Shareware）后，一般都有使用时间或功能上的限制。当过了共享软件的试用期后，必须到这个软件的公司去注册后方能继续使用。注册过程一般是用户把自己的私人信息（如用户名、电子邮件地址、机器特征码等）连同信用卡号码告诉软件公司，软件公司根据用户的信息利用预先写好的一个计算注册码程序（称为注册机，KeyGen）算出一个序列号，以电子邮件或传真等形式发给用户。用户在得到这个序列号后，按照注册需要的步骤在软件中输入注册信息和注册码，其注册信息的合法性由软件验证通过后，软件就会取消各种限制，如时间限制、功能限制等，而成为完全正式版本。软件在每次启动的时候，从磁盘文件或系统注册表中读出注册信息并对其进行检查。如果注册信息正确，则以完全正式版的模式运行，否则作为有功能限制或时间限制的版本来运行。注册的用户可以根据自己拥有的注册信息得到售后服务。当软件推出新版本后，注册的用户还可以向软件作者提供自己的注册信息来得到版本升级服务。这种保护实现起来比较简单，不需要额外的成本；用户购买也非常方便，因特网上80%的软件都是以这种方式保护的。

5.1.1　序列号保护机制

软件验证序列号的过程，其实就是验证用户名和序列号之间的数学映射关系。这个映射关系是由软件的设计者制定的，所以各个软件生成序列号的算法是不同的。显然，这个映射关系越复杂，注册码就越不容易被破解。根据映射关系的不同，程序检查注册码通常有如下4种基本的方法。

（1）以用户名等信息作为自变量，通过函数F变换之后得到注册码

将这个注册码和用户输入的注册码进行字符串比较或者数值比较，以确定用户是否为合法用户。其公式表示如下：

[image]

由于负责验证注册码合法性的代码是在用户的机器上运行的，因此用户可以利用调试器等工具来分析程序验证注册码的过程。上述方法中计算出来的序列号是以明文形式在内存中出现的，很容易在内存中找到它，从而获得注册码。这种方法在检查注册码合法性的同时，也在用户机器上再现生成注册码的过程（即在用户机器上执行了F函数）。实际上，这是非常不安全的。不论所采用的函数F有多么复杂，解密者只需

把F函数的实现代码从软件中提取出来，就可编制一个通用的计算注册码程序。由此可见，这种检查注册码的方法是极其脆弱的。解密者也可通过修改比较指令的办法，通过注册码检查。

（2）通过注册码验证用户名的正确性

软件作者在给注册用户生成注册码的时候，使用的仍然是下面的这种变换：

[image]

注意，这里要求F是个可逆变换。软件在检查注册码的时候则是利用F的逆变换F^{-1}对用户输入的注册码进行变换的。如果变换的结果和用户名相同，则说明是正确的注册码。即

[image]

可以看到，用来生成注册码的F函数未直接出现在软件代码中，而且正确注册码的明文也未出现在内存中，所以这种检查注册码的方法比第1种要安全一些。

破解这种注册码检查方法除了可以采用修改比较指令的办法之外，还有如下几种考虑：

• 由于F^{-1}的实现代码是包含在软件中的，所以可以通过F^{-1}来找出其逆变换即F函数，从而可以得到正确的注册码或者写出注册机；

• 给定一个用户名，利用穷举法找到一个满足式（5-2）的序列号，这只适用于穷举难度不大的函数；

• 给定一个序列号，利用式（5-2）变换得出一个用户名（当然这个用户名中一般包含不可显示字符），从而得到一个正确的用户名/序列号对。

（3）通过对等函数检查注册码

如果输入的用户名和序列号满足式（5-3），则认为是正确的注册码。采用这种方法，同样可以做到在内存中不出现正确注册码的明文。如果F_2是一个可逆函数，则本方法实际上是第2种方法的一个推广，解密方法也类似。

[image]

上面所说的3种检查序列号的方法中所采用的自变量都只有一个，自变量是用户名或序列号。

（4）同时采用用户名和序列号作为自变量，即采用二元函数

这种检查注册码的方法将采用如下的判断规则：当对用户名和序列号进行变换时，如果得出的结果和某个特定的值相等，则认为是合法的用户名/序列号对。

[image]

这个算法看上去相当不错，用户名与序列号之间的关系不再那么清晰了，但同时可能也失去了用户名与序列号的一一对应关系，软件开发

者自己都很有可能无法写出注册机，必须维护用户名称与序列号之间的唯一性，但这似乎不难办到，建个数据库就可以了。当然，也可根据这一思路把用户名称和序列号分为几个部分来构造多元的算法。

[image]

以上所说的都是序列号与用户名相关的情况，实际上序列号也可以与用户名根本不存在任何关系，这完全取决于软件作者的考虑。

由上可见，注册码的复杂性问题归根到底是一个数学问题。要想设计难以求逆的算法，要求软件作者有一定的数学基础。当然，即使检查注册码的算法再复杂，如果可执行程序可以被任意修改，解密者还是可以通过修改比较跳转指令来使程序成为注册版。所以，光有好的算法是不够的，还得结合软件完整性检查等其他方法。

5.1.2　如何攻击序列号保护

若要找到序列号，或者修改判断序列号之后的跳转指令，最重要的是要利用各种工具来定位判断序列号的代码段。

一种办法是通过跟踪输入注册码之后的判断，从而找到注册码。一般都是在一个编辑框中输入注册码，软件需要调用一些标准的API将编辑框中输入的注册码字符串拷贝到自己的缓冲区中。利用调试器提供的针对API设断点的功能，就有可能找到判断注册码的地方。这些常用的API包括GetWindowTextA(W)、GetDlgItemTextA(W)、GetDlgItemInt、hmemcpy（仅Windows 9x/Me）等。程序判断完注册码之后，一般显示一个对话框，告诉用户注册码是否正确，这也是一个切入点。显示对话框的常用API函数包括MessageBoxA(W)、MessageBoxExA(W)、DialogBoxParamA(W)、CreateDialogIndirectParamA(W)、DialogBoxIndirectParamA(W)、CreateDialogParamA(W)、MessageBoxIndirectA(W)、ShowWindow等。

另外一种办法就是跟踪程序启动时对注册码的判断，因为程序每次启动时都需要将注册码读出来加以判断，从而决定是否以注册版的模式工作。根据序列号的存放位置的不同，可以使用不同的API断点。如果序列号存放在注册表中，可以用RegQueryValueExA(W)；如果序列号存放在INI文件中，可以用GetPrivateProfileStringA(W)、GetProfileStringA(W)、GetPrivateProfileIntA(W)、GetProfileIntA(W)等函数；如果序列号存放在一般的文件中，可以用CreateFileA(W)、_lopen()等函数。

1．数据约束性的秘诀

这个概念是+ORC提出的，只限于用明文比较注册码的保护方式。在大多数序列号保护的程序中，那个真正的、正确的注册码会于某个时刻出现在内存中，当然它出现的位置是不定的，但多数情况下它会在一个范围之内，即存放用户输入序列号的内存地址±90h字节的地方。

数据约束性（Data constraint）或者“密码相邻性（Password proximity）”的依据就是加密者在编程的时候需要留意保护功能是否“工

作”，必须“看到”用户输入的数字、用户输入转换结果和真正密码之间的关系，这种联系必须经常地检查以调用这些代码。通常，它们会共同位于一个小的堆栈区域（注意：参数或局部变量通常都是存储在堆栈中的，而软件作者一般都使用局部变量存放临时计算出来的注册码），使得它们可以在同一个监视（Watch）窗口中看到，所以在大多数情况下，真正的密码会在离保存用户输入密码不远的地方露出马脚来。

例：运行第2章的TraceMe程序，输入用户名：pediy；序列号：12121212。单击“Check”按钮，TraceMe提示序列号错误，不要关闭此提示窗口。运行WinHex，单击菜单“Tools/RAM Editor”或按“Alt + F9”键打开内存编辑工具，单击“TraceMe”选项打开“Primary Memory”内存查看。按“Ctrl + F”键打开查找对话框，输入假的序列号“12121212”，在这附近会发现另一个字符串“2470”，这个就是真序列号，结果如图5.1所示。

[image]

图5.1　数据的约束性图

用OllyDbg也可实现这种查找功能。用OllyDbg加载TraceMe输入假序列号，单击“Check”按钮直到出现错误提示框。按“Alt + M”键打开内存窗口，在最上一行，按“Ctrl + B”键打开搜索框，搜索刚输入的序列号“12121212”，如图5.2所示。OllyDbg这个数据查找功能非常有用，可以在当前进程的整个内存映像里查找数据。

[image]

图5.2　在OllyDbg内存窗口中查找字符串

由于不少软件作者不了解这些基本的解密知识，算法上虽然下了很大工夫，但最后还是采用明码比较，这样导致一个会使用WinHex的普通用户都能找到其序列号。

2．hmemcpy函数（俗称万能断点）

函数hmemcpy是Windows 9x系统的内部函数，它的作用是将内存中的一块数据拷贝到另一个地方。由于Windows 9x系统频繁使用该函数处理各种字符串，因此用它作为断点很实用，它是Windows 9x/Me平台最常用的断点。在Windows NT/2000中没有这个断点，因为其内核和Windows 9x完全不同。

3．利用消息断点

许多序列号保护的软件都有一个按钮，当鼠标左键按下和释放时，将发送消息WM_LBUTTONDOWN(0201h)和WM_LBUTTONUP(0202h)，因此用这个消息下断很容易找到按钮的事件代码。

4．利用提示信息

大多数软件在设计时，都采用了人机对话方式。所谓人机对话，即软件在执行完某一段程序之后，便显示一串提示信息，以反映该段程序运行后的状态。例如，TraceMe实例输入假的序列号，如“序列号错误，再来一次！”。可以用OllyDbg、IDA、W32Dasm等反汇编工具查找相应字符串，定位到相关代码处。

用OllyDbg打开实例TraceMe，单击鼠标右键，执行“Search for/All referenced text strings（查找/所有参考文本字串）”命令，OllyDbg将列出程序中出现的字符串。但OllyDbg自带的这个功能对中文支持得不好，因此建议使用Ultra String Reference插件。安装好插件后，右键菜单中执行“Ultra String Reference/Find ASCII”，即列出中文字符串，双击相关字符串即可定位到所需代码上。

5.1.3　字符串比较形式

在序列号分析过程中，字符串处理是一个重点，必须掌握一定的分析技能。加密者为了有效防止解密者修改跳转指令，往往采取一些技巧，迂回比较字符串。

（1）寄存器直接比较

[image]

（2）函数比较1

[image]

在这种情况下，call一般是一个BOOL函数，其结果通过eax返回。分析时，关注该call里面返回时对eax处理的代码。例如call里面的代码：

[image]

（3）函数比较2

[image]

（4）串比较

[image]

5.1.4　注册机制作

软件开发结束后，作者本人很有必要先做攻击测试，找出弱点，避免犯一些低级错误。一般注册算法其实是一些极为简单的算法，基本都是明码的，或者明码相近的，如查表、异或、换位、移位、累加和等，算法实现都比较容易。

1．明码比较软件的攻击

只要正确的序列号在内存中曾以明码形式出现（不管比较时用不用

明码）都属于这一类。有些软件采取了一机一号的保护方式，即软件根据用户硬件等产生一个唯一的机器号，注册码与机器号对应有效地防止了序列号散发。如果是明码比较，攻击还是很容易的。能轻易实现这一目的，就是利用keymake软件，它拦截程序指令并将出现的明码以某种方式直接显示出来。

实例TraceMe的序列号是明码比较的，相关代码如下：

[image]

运行keymake后，单击菜单“其他/内存注册机”，打开如图5.3所示的界面。

[image]

图5.3　keymake的内存注册机

具体操作步骤如下：

① 单击“浏览”按钮，打开目标程序TraceMe.exe。

② “内存方式”选择寄存器，本例是EBP，即序列号保存在EBP所指向的内存地址中。

③ 中断地址列表：

• 中断地址，目标程序明码比较指令地址；

• 次数，在此地址中断的次数；

• 指令，此处指令指机器码第一个字节；

本例，keymake需要拦截TraceMe两次，第一次是4011E5h的call；进去后，再次拦截40138Dh地址，以便查看EBP指向的字符串。设置信息见图5.3 。

按上述设置好后，单击“生成”按钮就可生成一个注册机，使用时该注册机和目标程序放到同一目录里。运行时，注册机装载目标程序，在指定地址处插入一个INT 3指令，目标程序会在此中断，然后注册机将内存或寄存器值读出，再恢复原程序指令。TraceMe被装载后，输入用户名，单击“Check”按钮，注册机将跳出一个窗口告知正确的序列号（见图5.4）。

[image]

图5.4　生成的序列号

[image]注意：杀毒软件会将keymake生成的文件报告为木马或病毒，读者可以参考“补丁技术”一章，自己编程实现内存注册机。

2．非明码比较

实例Serial.exe是通过对等函数检查序列号。如果输入的用户名和序列号满足计算式：F_1（用户名）= F_2（序列号），则认为是正确的序列号。采用这种方法，可以做到在内存中不出现明码。

单击实例Serial.exe菜单“Help/Register”打开注册窗口，这个窗口是

用DialogBoxParamA函数建立，EndDialog函数关闭的。可以用函数GetDlgItemTextA、EndDialog等设断拦截。由于程序是先关闭对话框才开始比较序列号，因此要在系统里走一段才能回到Serial.exe程序领空。也可以直接从提示信息切入找到关键点。用OllyDbg装载Serial.exe后，输入姓名pediy，序列号1234，单击“OK”按钮后，将跳出“Incorrect!,Try Again”提示窗口，记下这串字符。用鼠标右键打开“Search for/All referenced text strings”交叉参考字符串窗口，找到“Incorrect!,Try Again”，双击就可来到关键代码处。很明显401228h这段代码处理输入的字符串“pediy”，在此按F2键设断，重新来一次，运行程序单步分析如下。

[image]

call 0040137E函数内部代码：

[image]

[image]

[image]

上面的代码是计算k1 = F_1（用户名），用C语言来描述，代码如下：

[image]

call 004013D8函数内部代码：

[image]

[image]

上面的代码是计算k2 = F_2（序列号），用C语言来描述，代码如下：

[image]

只要满足关系式：k1 = k2，注册就成功。写注册机就要对F_1或F_2函数逆变换，若F_1和F_2都不可逆，只能用穷举法。如要从用户名算出正确的序列号，只要写出F_2逆函数即可：k1 = F_2^{-1}（序列号）。但求逆F_2函数有多个解，比较复杂，幸运的是，k1的结果是个十六进制数，这样问题就简单了，F_2函数功能可看作是将输入的十进制数转换成十六进制数。

注册机如下：

[image]

算法求逆是有难度的，这需要一定的编程基本功。常见的加密配对

指令有xor/xor、add/sub、inc/dec、rol/ror等，这些指令就是一条可以加密，另一条指令可以解密。

还有一种写注册机方法是不分析其运算过程，用OllyDbg的Asm2Clipboard插件、IDA等工具可直接将序列号算法的汇编代码提取出来，再嵌入到高级语言中。这个方法的优点是不用理解算法实现的细节，只需要将汇编代码嵌入到注册机里即可。比如，F_1函数就是这种情况，将40137Eh到4013D6h之间的汇编代码转换成asm文件格式，然后嵌入到高级语言中调用。代码转换中要注意堆栈平衡、数据进制、汇编语法格式、字符串引用等。下面就是直接提取汇编代码内嵌到VC里的代码：

[image]

[image]

5.2　警告（Nag）窗口

Nag的本义是烦人的意思。Nag窗口是软件设计者用来不时提醒用户购买正式版本的窗口。软件设计者可能认为，当用户忍受不了试用版中的这些烦人的窗口时，就会考虑购买正式版本。它可能会在程序启动或退出时弹出来，或者在软件运行的某个时刻随机或定时地弹出来，确实比较烦人。

去除警告窗口常用的3种方法是：修改程序的资源、静态分析及动态分析。

使用资源修改工具去除警告窗口是个不错的方法，可以将可执行文件中的警告窗口的属性改成透明、不可见，这样就变相地去除了警告窗口。

若要完全去除警告窗口，只需找到创建此窗口的代码，跳过即可。显示窗口的常用函数有MessageBoxA(W)、MessageBoxExA(W)、DialogBoxParamA(W)、ShowWindow、CreateWindowExA(W)等。然而，某些警告窗口用这些断点不管用，可试一试利用消息设置断点，一般都能拦截下来。

实例Nag.exe是一个显示警告窗口的程序，调用DialogBoxParamA函数来显示资源中的对话框。由于Nag.exe是调用资源来显示对话框的，因此用eXeScope或Resource Hacker打开它，显示警告窗口的资源如图5.5所示。

[image]

图5.5　查看对话框资源

启动画面窗口的ID号是121，换成十六进制就是79h。用W32Dasm打开Nag.exe，打开对话框参考，里面的“Dialog:DialogID_0079”项就是Nag.exe刚运行时跳出的对话框，双击此项可以来到相关代码处。W32Dasm里的具体代码如下：

[image]

DialogBoxParam一般和EndDialog配对使用，前者是打开对话框，后者是关闭对话框。因此，不能简单地将DialogBoxParam屏蔽掉。DialogBoxParam原型如下：

[image]

从上面函数看出，lpDialogFunc参数很重要，DialogBoxParam函数将跳到其指向的地址执行，对lpDialogFunc参数（此处为4010C4h）设断。中断后代码如下：

[image]

原来主程序也是用DialogBoxParam函数显示的，因此有两种改法：

（1）跳过警告窗口那段代码

将“00401051 push 00000000”一句改成：“00401051 jmp 4010E5”。

修改时在OllyDbg里键入正确的代码，选择修改后的代码，执行右键菜单中的“复制到可执行文件”功能，即可将修改保存到磁盘文件中。

（2）将两个DialogBoxParam函数参数对换

DialogBoxParam函数有两个参数很重要，一个是主对话框处理函数指针，另一个是对话框的ID号。这种方法的思路是将主窗口的这两个参数放到警告窗口的DialogBoxParam函数上。修改如下：

[image]

在另外一些情况下，对话框不是以资源形式存在的，而常用断点又拦不下来，这时可试试消息断点，如WM_DESTROY等。

5.3 时间限制

时间限制程序有两类：一类是每次运行多少时间；另一类是每次运行时间不限，但是有个时间段限制，如用30天等。

5.3.1 计时器

这类程序每次运行时都有时间限制，例如运行10分钟或20分钟就停止，必须重新运行该程序才能正常工作。这些程序里面自然有个计时器统计程序运行的时间。那么如何实现计时器呢？在DOS系统下，应用程序可以通过接管系统的计时器中断（一般为int 8h或int 1Ch）维护一个计时器，它能每55毫秒发生一次（18.2次每秒）。在Windows下，使用计时器有如下几种不同的选择。

1．使用SetTimer()函数

应用程序可在初始化时调用这个API函数来向系统申请一个计时器，并且指定计时器的时间间隔；还可提供一个处理计时器超时的回调函数。当计时器超时时，系统将会向申请该计时器的窗口过程发送消息

WM_TIMER，或者调用程序所提供的那个回调函数。

该函数的原型如下：

[image]

各参数的含义如下。

• hWnd：窗口句柄，当计时器时间到时时，系统将向这个窗口发送WM_TIMER消息。

• nIDEvent：计时器标识。

• uElapse：指定计时器时间间隔，以毫秒为单位。

• TIMERPROC：回调函数。当计时器超时时，系统将调用这个函数。如果本参数为NULL，当计时器超时时将向相应的窗口发送WM_TIMER消息。这个回调函数的原型为：

[image]

由于SetTimer()是以Windows消息的方式工作的，所以其精度有一定的限制，但对于做软件保护来说已经够用。当程序不再需要计时器时，可以调用KillTimer()来销毁计时器。

2．使用高精度的多媒体计时器

多媒体计时器的精度最高可以达到1毫秒。应用程序可以通过调用timeSetEvent()启动一个多媒体计时器。该函数的原型如下：

[image]

3．GetTickCount()

Windows提供了API函数GetTickCount()。该函数返回的是系统自成功启动以来所经过的毫秒数。将该函数的两次返回值相减，就可知道程序已经运行了多长时间。这个函数的精度取决于系统的设置。实际上，也可以在高级语言里面利用各自开发库中所提供的函数来实现计时，比如在C语言中就可使用time()函数获得系统时间。

4．timeGetTime()

多媒体计时器函数timeGetTime()也可以返回Windows自启动后所经过的时间，以毫秒为单位。一般情况下，不需要使用高精度的多媒体计时器，精度太高对系统性能也会有影响。

5.3.2 时间限制

这类保护的软件一般都有时间段的限制，例如试用30天等。当过了共享软件的试用期后，就不予运行。只有向软件作者付费注册之后才能得到一个无时间限制的注册版本。这种保护的实现方式大致如下。

首先在安装软件的时候由安装程序取得当前系统日期，或者主程序在第一次运行的时候获得系统日期，并且将其记录在系统中的某个地方；可能记录在注册表的某个不显眼的位置，也可能记录在某个文件或扇区中。这个时间统称为软件的安装日期。

程序在每次运行的时候都要取得当前系统日期，且将其与记录下来的那个安装日期进行比较，当其差值超出允许的天数（比如30天）时就停止运行。

可见，这种日期限制的机理很简单。但是在实现的时候，如果对各种情况处理得不够周全，就很容易被绕过，比如在过期之后简单地把机器时间调回去，软件又可以正常使用了。

如果考虑得比较周全，软件最少要保存两个时间值，一个就是上面所说的安装时间，这个时间可由安装程序在安装软件的时候记录，也可以在软件第一次运行的时候记录（即软件发现该值不存在时就将当前日期作为其值记录下来）。为了增加解密难度，最好把这个时间在不同的地方多存放几份，否则解密者可以通过RegMon、FileMon等监视工具轻易地找到存放该值的地方，然后删除该键值，这样又可以正常使用软件了。

另外一个时间值就是软件最近一次运行的日期，这是防止用户将机器日期改回去而设的。软件每次退出的时候都要将该日期取出来与当前日期相比较，如果当前日期大于该日期，则用当前日期替换掉该值，否则保持该值不变。同时，软件每次启动的时候要把该值读出来与当前日期进行比较，如果该值大于当前系统日期，则说明用户把机器时间改回去了，可以拒绝运行。

取得时间的API函数一般有GetSystemTime、GetLocalTime和GetFileTime。软件作者可能不直接使用上面的函数来获得系统时间，比如采用高级语言中封装好的类来操作系统时间等。这些封装好的类实际上也是调用上面的函数。解密者在采用动态跟踪方法破解这种日期限制时，最常用的断点也是这几个。

还有一种比较方便地获得当前系统日期的办法，就是读取需要频繁修改的系统文件（比如Windows注册表文件user.dat、system.dat等）的最后修改日期，利用FileTimeToSystemTime()将其转换为系统日期格式，从而得到当前系统日期。

需要指出的是，采用日期限制的软件必须能防RegMon、FileMon之类的监视软件，否则很容易被找到日期的存放位置。

5.3.3　拆解时间限制保护

实例Timer.exe程序用了SetTimer函数计时，每次运行20秒。其运行原理是，先用SetTimer(hwnd,1,1000,NULL)设置一个计时器，时间间隔是1000毫秒，这个函数每秒发一次WM_TIMER消息。当应用程序收到消息时，将执行下面的语句：

[image]

因此，可以用SetTimer函数设断拦截。

[image]

去除时间有如下两种方法。

方法一：直接跳过SetTimer函数，不产生WM_TIMER消息。

来到地址4010C6，输入修改指令“jmp 4010D6”。

方法二：利用WM_TIMER消息。

查VC的头文件WINUSER.H，得知：#define WM_TIMER 0x0113。

在W32Dasm里查找113字串（当然，实际中有可能用其他形式检查是否为113），如下所示：

[image]

因此，只要修改00401184一行就能取消时间限制，用两个字节替换掉，如9090或eb00。

另外，辅助工具变速齿轮可加快和缩短应用程序的时间，一般用来配合动态分析。例如，某软件运行一小时后才退出，此时可以用变速齿轮加速时间，几分钟后，软件就认为到了一小时而退出，从而更方便调试程序。

5.4 菜单功能限制

这类程序一般是Demo版，其菜单或窗口中的部分选项是灰色，无法使用。这种功能受限制的程序一般分成两种：第一种是试用版和正式版的软件完全分开的两个版本，被禁止的功能在试用版的程序中根本没有相应的程序代码，这些代码只有在正式版中才有，而正式版是无法免费下载的，只有向作者购买。对于这种程序，解密者要想在试用版中使用和正式版一样的功能几乎是不可能的，除非自己向可执行程序中添加相应的代码。第二种是试用版和注册版为同一个文件，没有注册的时候按照试用版运行，禁止用户使用某些功能。一旦用户注册之后就以正式版模式运行，用户可以使用全部功能。可见，被禁止的那些功能的程序代码其实是存在于程序之中的，解密者只要通过一定的方法恢复被限制的功能，就能使该Demo软件与正式版一样。对比一下就知道，前一种显然更好，因为它使得破解难度大大增加。如果采用功能限制的保护方式，强烈建议使用前一种方式。

5.4.1 相关函数

软件将菜单或窗口变灰或变为不可用，一般采用下面的几个函数。

1．EnableMenuItem

允许或禁止指定的菜单条目。原型如下：

[image]

各参数的含义如下。

• hMenu：菜单句柄。

• uIDEnableItem：欲允许或禁止的一个菜单条目的标识符。

• uEnable：控制标志，有MF_ENABLED（允许，0h）、MF_GRAYED（灰化，1h）、MF_DISABLED（禁止，2h）、MF_BYCOMMAND和MF_BYPOSITION。

返回值：返回菜单项以前的状态，如果菜单项不存在，就返回

FFFFFFFFh。

2．EnableWindow

允许或禁止指定窗口。原型如下：

[image]

各参数的含义如下。
• hWnd：窗口句柄。
• bEnable：TRUE允许，FALSE禁止。
返回值：非0表示成功，0表示失败。

5.4.2　拆解菜单限制保护

名称：EnableMenu，文件：光盘\chap05\
这个程序是用EnableMenuItem函数禁止菜单，其关键代码如下：

[image]

uEnable控制标志为0时，恢复菜单的功能，具体操作如下：
将“004011E3 push 00000001”改成“push 0”。

5.5　KeyFile保护

KeyFile是一种利用文件来注册软件的保护方式。KeyFile一般是一个小文件，可以是纯文本文件，也可以是包含不可显示字符的二进制文件。其内容是一些加密过或未加密的数据，其中可能有用户名、注册码等信息，文件格式则由软件作者自己定义。试用版软件没有注册文件。当用户向作者付费注册之后，会收到作者寄来的注册文件，其中可能包含用户的个人信息。用户只要将该文件放入指定的目录，就可以让软件成为正式版。该文件一般放在软件的安装目录中或系统目录下。软件每次启动时，从该文件中读取数据，然后利用某种算法进行处理，根据处理的结果判断是否为正确的注册文件。如果正确，则以注册版模式运行。

在实现这种保护的时候，建议软件作者采用稍大一些的文件作为KeyFile，一般在几KB左右。其中可以加入一些垃圾信息以干扰解密者的企图；对于注册文件的合法性检查也可以分成几部分，分散在软件的不同模块中进行判断；对注册文件内的数据处理也尽可能地采用复杂的运算，而不要使用简单的异或运算。这些措施都可增大解密难度。和注册码一样，也可以让注册文件中的部分数据和软件中的关键代码或数据发生关系，使得软件无法被暴力破解。

5.5.1　相关API函数

由于Key File是一个文件，因此所有有关Windows文件操作的API函数都可作为动态跟踪破解的断点。这类常用的文件函数如表5-1所示。

表5-1　与KeyFile相关的函数

[image]

各API函数的具体含义参考MSDN或相关API文档。

5.5.2 拆解KeyFile保护

名称：PacMe，文件：光盘\chap05\

1．拆解**KeyFile**的一般思路

① 先用FileMon等工具监视软件对文件的操作，以找到KeyFile的文件名。

② 伪造一个KeyFile文件。用十六进制工具编辑和修改KeyFile，普通的文本编辑工具不太适合。

③ 在调试器里用CreateFileA函数设断查看其打开文件名指针，并记下返回的句柄。

④ 用ReadFile设断，分析传递给ReadFile的文件句柄和缓冲区地址。文件句柄一般和第③步的相同（若不同，则说明不是读该KeyFile，此外也可用条件断点）。缓冲区地址则是非常重要的，因为读进来的重要数据放在这里。对缓冲区中存放的字节设内存断点，监视读进来的KeyFile的内容。

⑤ 当然上述步骤只是大概轮廓，有的程序判断KeyFile时还会先判断文件大小和属性、移动文件指针等。

总之，分析KeyFile取决于对Win32 File I/O API的熟悉程度，也就是API编程的水平。

2．监视文件的操作

PacMe的注册信息放在某一文件中，可以用文件监视工具得到答案。FileMon是一个不错的选择，使用时，建议设置一下过滤器。所谓过滤器，其实就是一组条件，这组条件用来限制FileMon什么该显示，什么不该显示，如图5.6所示。

[image]

图5.6 过滤器配置对话框

设置好后，按“Ctrl + E”键捕捉事件，按“Ctrl + X”键清除所有的记录。FileMon按时间的顺序记录系统中发生的各种文件访问事件。

3．分析过程

除了用FileMon监视文件获得KeyFile文件名，也可以直接对文件相关函数设断获得KeyFile相关信息。用OllyDbg装载PacMe后，按F9键运行PacMe。用CreateFileA设断，单击PacMe的“Check”按钮中断如下：

[image]

OllyDbg直接把CreateFileA读取的文件名显示出来。很明显，KeyFile名为KwazyWeb.bit。用十六进制工具伪造一个KeyFile，建议内

容为一些有规律的数字，如12345…等，以便跟踪时有利于分析。重新运行程序，PacMe将打开KwazyWeb.bit文件，读取数据进行计算比较。代码如下：

[image]

[image]

再来分析一下验证的核心代码：

[image]

[image]

[image]

[image]

[image]

这是一个标准的迷宫，从“C”开始，一共走18次，每次可以走4步（18次大循环和4次小循环）。碰到“*”就中断，直到遇见“X”时就注册成功。而且路线非常清楚，顺着“.”走。按照上面的程序分析，“0”代表↑，“1”代表→，“2”代表↓，“3”代表←，看着图一步步向前进，就可以得到一系列数据（见图5.7）。

[image]

图5.7　PacMe的迷宫路线

上面是四进制数，转换成十六进制数为：

A9 AB A5 10 54 3F 30 55 65 16 56 BE F3 EA E9 50 55 AF

然后，程序通过用户名计算一数据，再与上面的十六进制数异或。

在此以用户名pediy推出KeyFile，pediy的十六进制代码是：7065646979。KeyFile由三部分组成，如图5.8所示。

[image]

图5.8　PacMe的KeyFile内容

① 先计算pediy字符的和：70 + 65 + 64 + 69 + 79 = 21B，取低8位为：1B；

② 然后用1B依次与“A9 AB A5 10 54 3F 30 55 65 16 56 BE F3 EA E9 50 55 AF”异或，结果是“B2 B0 BE 0B 4F 24 2B 4E 7E 0D 4D A5 E8 F1 F2 4B 4E B4”。

5.6　网络验证

网络验证是目前流行的一种保护技术，其优点是可以将一些关键数

据放到服务器上，软件运行时，必须从服务器取得这些数据才能正确运行。拆解的思路是拦截服务器返回的数据包，分析程序是如何处理数据包的。

5.6.1　相关函数

当一个连接建立以后，就可以传输数据了，常用的传送数据的函数有send和recv两个Socket函数，另外还有微软的扩展函数WSASend和WSARecv。

1．send函数

客户程序一般用send函数向服务器发送请求，而服务器则通常用send函数来向客户程序发送应答。

[image]

2．recv函数

不论是客户还是服务器应用程序都用recv函数从TCP连接的另一端接收数据。

[image]

5.6.2　网络验证破解一般思路

如果网络验证的数据包内容固定，可以将数据包抓取，写个本地服务端模拟服务器；如果验证的数据包内容不固定，则必须分析出其结构，找出相应的算法。

实例CrackMeNet.exe是一款网络验证实例，CrackMeNetS.exe是服务端，并提供了一组正确的登录账号。实际过程中，服务端是接触不到的，读者必须从客户端入手，并利用一组正确的账号，击破这个网络验证保护。

1．分析发送的数据包

建议用IDA与OllyDbg一起来分析，IDA能正确识别出C函数，方便分析。OllyDbg加载客户端后，用send函数设断，输入正确的账号与口令，单击“Register”按钮，中断并回到当前领空。代码如下：

[image]

send函数将把Data缓冲区中的数据发送到服务端，这里查看Data数据，发现是加密的。在IDA中向前查看代码，再结合OllyDbg分析，这段代码功能如下：

[image]

[image]

[image]

原来客户端将输入的Name及Key按图5.9所示的格式处理，并异或加密发送给服务端。

[image]

图5.9　发送的数据包

2．分析接收的数据包

服务端接收到数据后，经过计算，将包括正确的数据包返回给客户端。客户端程序用recv接收数据，相关代码如下：

[image]

[image]

上面这段代码接收到数据，并进行解密，解密后的数据存放在41AE68h～41AEC1h这段空间，如图5.10所示。

[image]

图5.10　解密后的数据包

接下来程序会从41AE68h～41AEC1h读取所需要的字节，因此只要对这段数据下内存读断点，很容易定位到相关代码处。但在实际应用中，程序读取这部分数据可能比较隐蔽，比如运行一段时间再比较，或用到某功能后再比较等。因此有可能遗漏相关的读取代码。

本实例是用全局变量构建缓冲区的，由于是以Debug编译的程序，程序里会直接用如下指令读取缓冲区数据。

[image]

一个简单并且有效的办法，是在整个代码里搜索访问41AE68h～41AEC1h这段缓存区的mov指令。这时，IDA强大就体现出来了，用IDA打开如下脚本，就可将读取指定内存的代码列出了。

[image]

[image]

先用“File/IDC File”打开getasm.idc脚本，按“Shift＋F2”键打开IDC命令行，键入“Getasm(0x401000,0x40F951,0x41AE68,0x0041AEC1);”，如图5.11所示。

[image]

图5.11　IDC命令行输入命令

这个脚本就将整个程序访问缓冲区所有指令列出了。打开c:\code.txt文件，内容如下：

[image]

本例将缓冲区放全局变量，一般情况下，缓冲区一般是放局部变量的，其访问缓冲区数据的指令如下，其中r/m32表示32位寄存器。

[image]

本节提供的getasm.idc脚本也支持上述指令，只需要确实指令的范围（这里是n的范围），即可得到正确的访问缓冲区的指令。在实际情况中，访问同一数据块可能用不同指针，也可能将数据块复制到其他地方再读取。

3．解除网络验证

发送与接收的封包都已分析出，比较省事的解决方法是写个服务端，模拟服务器接收和发送数据。如果软件是用域名登录服务器的，可以修改hosts文件，使得域名指向本地127.0.0.1。如果直接用IP连接服务器，可以用inet_addr或connect等设断，修改IP地址为本地。或使用代理软件将IP用代理指向本地。

除了写服务端外，也可直接修改客户端程序，将封包中的数据整合进去，下面主要讲述一下这种改法。

将CrackMeNet.exe复制一份，用OllyDbg打开，然后将开始截取的正确数据粘贴到41AE68h～41AEC1h这段地址处。

第一步，将发包功能（send）去除，再将随机数读取到41AE76h处。

[image]

第二步，将recv函数去除，并跳过数据解密代码，修改代码：

[image]

经过这样处理，再运行实例，单击“Register”按钮，跳出一对话框提示“Error:Connection failed.”，直接强行跳过即可。修改代码如下：

[image]

从以上分析可以看出，网络验证关键就是数据包分析。数据包可以用一些工具辅助，如WPE、iris等。如果数据包加密或要彻底分析数据包处理过程，必须用发送/接收函数设断，跟踪程序对数据包的处理。

5.7　CD-Check

一些采用光盘形式发行的应用软件和游戏在使用时需要检查光盘是否插在光驱中，如果没有，则拒绝运行。这是为了防止用户将软件或游

戏的一份正版拷贝安装在多台机器上并同时使用。这和DOS时代的钥匙盘保护是类似的，虽然能在一定程度上防止非法拷贝，但也给正版用户带来了一些麻烦；一旦光盘被划伤，用户就无法使用软件了。这里将介绍常见的光盘检测的实现方式，以及如何去除光盘检测的基本知识。其他一些光盘专业保护软件，如SafeDisc等很复杂，本节不作讲述。

最简单也最常见的光盘检测就是程序在启动时，判断光驱中的光盘上是否存在特定的文件。如果不存在，则认为用户没有使用正版光盘，拒绝运行。在程序运行的过程中，一般不再检查光盘的存在与否。Windows下的具体实现一般是这样的：先用GetLogicalDriveStrings()或GetLogicalDrives()得到系统中安装的所有驱动器的列表，然后再用GetDriveType()检查每个驱动器；如果是光驱，则用CreateFileA()或FindFirstFileA()函数检查特定的文件存在与否，并且可能进一步地检查文件的属性、大小、内容等。

这种光盘检查是比较容易被破解的，解密者只要利用上述函数设置断点，找到程序启动时检查光驱的地方，然后修改判断指令就可以跳过光盘检查。

上述保护的一种增强类型，就是把程序运行时所需要的关键数据放在光盘中。这样，即使解密者能够强行跳过程序启动时的检查，由于没有使用正版光盘，也就没有程序运行时所需要的关键数据，程序自然崩溃，这样可以在一定程度上起到防破解的作用。

对付上述这种增强型光盘保护还是有办法的，比如简单地利用刻录、拷贝工具将光盘复制多份。也可采用虚拟光驱程序来模拟正版光盘。常用的虚拟光驱程序有Virtual CD、Virtual Drive、Daemon Tools等，尤其值得一提的是Daemon Tools，它不仅是免费的，而且能够模拟一些加密光盘。这些光盘加密工具一般都在光轨上做文章，比如做暗记等。有的加密光盘可用工作在原始模式（Raw mode）的光盘拷贝程序来原样拷贝，比如用Padus公司的DiscJuggler和Elaborate Bytes公司的CloneCD等拷贝工具。对光盘加密感兴趣的读者可以查阅ISO9660标准协议。

5.7.1 相关函数

1．GetDrivetypeA(W)

获取磁盘驱动器类型。

[image]

2．GetLogicalDrives

获取逻辑驱动器符号。

[image]

3．GetLogicalDriveStrings

获取当前所有逻辑驱动器的根驱动器路径。

[image]

4．GetFileAttributes A(W)

判断指定文件的属性。

[image]

5.7.2　拆解光盘保护

名称：CD_Check，文件：光盘\chap05\

这个程序先用GetDriveTypeA检测文件是否在光驱里，再用CreateFileA尝试打开光盘文件。如果存在，则成功。

[image]

5.8　只运行一个实例

Windows是一个多任务的操作系统，应用程序可以多次运行以形成多个运行实例。但有时出于某种考虑（比如安全性），要求程序只能运行一个实例。

5.8.1　实现方法

只运行一个实例的实现方法有多种，在此只列出几种常见的方法。

1．查找窗口法

这是最为简单的一种方法。在程序运行前，用Find Window、GetWindowText函数查找具有相同窗口类名和标题的窗口。如果找到了，就说明已经存在一个实例。

[image]

程序中代码如下：

[image]

2．使用互斥对象

尽管互斥对象通常用于同步连接，但用在这个地方也是非常方便的。一般是用CreateMutex函数实现，它的作用是创建有名或者无名的互斥对象。

[image]

程序中代码一般如下：

[image]

3．使用共享区块

创建一个共享区块（Section），该节拥有读取、写入和共享保护属性，让多个实例共享同一内存。将一个变量放到该区块中作为计数器，该应用程序的所有实例均可以共享该变量，以便通过该变量得知有没有实例在运行。

5.8.2　实例

“序列号保护方式”一节中的Serial.exe只能同时运行一个实例，该程序利用了FindWindow函数查找指定字串来确定程序是否运行了。对付这类保护最有效的办法是修改应用程序的窗口标题，修改FindWindow返回值也能取消其限制。

[image]

5.9　常用断点设置技巧

设置正确的断点，对调试软件是非常重要的。如果掌握一些Win32编程，对设置合适的断点是非常有帮助的。下面分门别类整理出常用的断点集合，方便查阅。

[image]

第6章　加密算法①

现有的序列号加密算法大多是软件开发者自行设计的，大部分相当简单，而且有些算法作者虽然下了很大的工夫，却往往达不到所希望的效果。其实，有很多成熟的算法可用，特别是密码学中的一些强度比较高的算法，比如RSA、BlowFish、MD5等。这些算法在因特网上有大量的源码或编译好的库（当然这些库中可能会有些漏洞），可以直接加以利用，所要做的只是利用搜索引擎找到它们并将其嵌入自己的程序中。应当指出，即使这些算法的强度很高，但是使用方法也要得当，否则效果就和普通的四则运算效果没有什么两样了，很容易被解密者算出注册码或者写出注册机来。

6.1　单向散列算法

单向散列函数算法也称Hash（哈希）算法，是一种将任意长度的消息压缩到某一固定长度（消息摘要）的函数（该过程不可逆）。Hash函数可用于数字签名、消息的完整性检测、消息起源的认证检测等。常见的散列算法有MD5、SHA、RIPE-MD、HAVAL、N-Hash等。

在软件的加密保护中，Hash函数是经常用到的加密算法。但是，由于Hash函数为不可逆算法，所以软件只能使用Hash函数作为一个加密的中间步骤。例如，对用户名做一个Hash变换，将这个结果再进行一个可逆的加密变换（如对称密码），变换结果为注册码。从解密角度来说，一般不必了解Hash函数的具体内容（变种算法除外），只要能识别出是何种Hash函数就可以了，然后直接套用相关算法源码实现。

6.1.1　MD5算法

MD5消息摘要算法（Message Digest Algorithm）是由R.Rivest所设计的。它对输入的任意长度的消息进行运算，产生一个128位的消息摘要。近几年来，随着穷举攻击和密码分析的发展，应用最为广泛的MD5算法已经不再那么流行了。

1．算法原理

（1）数据填充

填充消息使其长度与448模512同余（即长度≡448 mod 512）。也就是说，填充后的消息长度比512的倍数仅小64位的数。即使消息长度本身已经满足上述长度要求，仍然需要填充。

填充方法是附一个1在消息后面，然后用0来进行填充，直到消息的长度与448模512同余。至少填充1位，至多填充512位。

（2）添加长度

在第一步的结果之后附上64位的消息长度。如果填充前消息的长度大于2^{64}，则只使用其低64位。

这时，在添加完填充位和消息长度之后，最终消息的长度正好就是512的整数倍了。

令M[0...N-1]表示最终的消息，其中N是16的倍数。

（3）初始化变量

用到4个变量（A，B，C，D）来计算消息摘要。这里A，B，C，D分别都是一个32位的寄存器。这些寄存器以下面的十六进制数值来初始化：A = 01234567h，B = 89abcdefh，C = fedcba98h，D = 76543210h。

并且在内存中是以低字节在前的形式来存储的，即如下格式：

[image]

（4）数据处理

以512位分组为单位处理消息，首先定义4个辅助函数，每个都是以3个32位双字作为输入，输出一个32位双字。

$F(X,Y,Z)=(X \& Y)|((\sim X) \& Z)$

$G(X,Y,Z)=(X \& Z)|(Y \& (\sim Z))$

$H(X,Y,Z)=X \wedge Y \wedge Z$

$I(X,Y,Z)=Y \wedge (X|(\sim Z))$

其中，&是与操作，|是或操作，~是非操作，^是异或操作。

这4轮变换是对进入主循环的512位消息分组的16个32位字分别进行如下操作：将A，B，C，D的副本a，b，c，d中的3个经F，G，H，I运算后的结果与第4个相加，再加上32位字和一个32位字的加法常数，并将所得之值循环左移若干位，最后将所得结果加上a，b，c，d之一，并回送至A，B，C，D，由此完成一次循环。

所用的加法常数由这样一张表T[i]来定义，其中i为1至64之中的值，T[i]等于4294967296乘以abs(sin(i))所得结果的整数部分，其中i用弧度来表示。这样做是为了通过正弦函数和幂函数来进一步消除变换中的线性。

接着进行如下的操作（伪码表示）：

[image]

[image]

（5）输出

当所有的512位分组都运算完毕后，ABCD的级联将被输出为MD5散列的结果。以上是MD5算法的简单描述，更为详细的实现程序，请参考光盘中的源代码。

2．MD5在加解密中的应用

MD5将任意长度的字符串变换成一个128位的大整数，并且是不可逆的。换句话说，即便MD5算法的源代码满天飞，使得任何人都可以了解MD5的详尽算法描述，但是绝对没有任何人可以将一个经由MD5算法加密过的字符串还原回原始的字符串。

在实际过程中，若单向散列算法运用不当，是没有什么保护效果的。看一看下面使用MD5判断注册码的伪码：

[image]

由于正确的序列号以明文形式出现在内存中，因此解密者很容易找到正确的注册码，写注册机只要识别出程序采用的是MD5算法就够了。遗憾的是，不少软件作者都偏爱这种判断注册码的方法。

MD5代码的特点非常明显，跟踪时很容易发现。如果软件采用MD5算法，在数据初始化的时候必然会用到上面提到的4个常数（A，B，C，D）。实际上，像KANAL这样的算法分析工具不是通过上面的这4个常数来鉴别MD5，而是通过识别具有64个常量元素的表T来确定是不是MD5算法的。对于变形的MD5算法，常见的有三种情况：一是改变初始化时所用到的4个常数；二是改变填充的方法；三是改变Hash变换的处理过程。在解密时，只要跟踪到以上提到的这些点，然后相应地对MD5的源代码进行修改，就可以实现相应的注册机制了。

3．实例分析

以光盘中的MD5KeyGenMe为例，讲述MD5算法在软件保护中的应用。先用PEiD插件Krypto ANALyzer分析光盘中的MD5KeyGenMe.exe，得知该KeyGenMe含有MD5的迭代（压缩）常数，猜测可能使用了MD5算法，如图6.1所示。

[image]

图6.1　用PEiD的插件扫描目标程序的加密算法

用OllyDbg打开实例MD5KeyGenMe.exe，按F9键运行KeyGenMe。在命令栏中输入bp GetDlgItemTextA，分别输入Name: pediy和Serial Number: 0123456789ABCDEF，单击“Check”按钮，程序将中断在GetDlgItemTextA函数开始处，单步调试来到实例代码中。

[image]

在上面的这段代码中，004011DB处的“push ecx”，ecx中即为MD5 Context地址，将用来存储MD5的结构。跟进004012B0可以看到如下代码：

[image]

根据代码中的4个常数，很明显这是在进行MD5初始化。但是，仅仅根据这一点还不能完全认定这就是MD5，需要进一步的判定。接下来进行信息的填充，首先把Name填充到Context中，接着填充一固定的字符串“www.pediy.com”，作为附加消息。在填充的过程中，若信息的长度满足一定的条件，将执行MD5 Transform操作，即对消息进行处理。代码如下：

[image]

对于在地址00401441处所出现的D76AA478h，是前面所提到的

MD5中的正弦函数表中的元素之一，在后续代码中出现了表中剩下的其他元素，并且根据这些代码所特有的操作特征（MD5的F，G，H，I函数），就可以断定是MD5无疑了。

相关代码如下：

[image]

经过上面的这个call之后，最终的散列值将保存在edx (0012F824)中，如下：

[image]

上面这段代码计算出来的MD5值，相当于将输入的Name字符与“www.pediy.com”连接，再计算其MD5值。

[image]

[image]

至此，该KeyGenMe的注册验证流程大致分析完了，可以根据此流程做出注册机。注册机源代码见本书光盘。

6.1.2　SHA算法

安全散列算法（Secure Hash Algorithm）简称SHA，有SHA-1、SHA-256、SHA-384和SHA-512几种，分别产生160位、256位、384位和512位的散列值。

下面以SHA-1为例，对SHA系列散列函数作简要介绍。

1．算法描述

SHA-1是一种主流的散列加密算法，其设计是基于和MD4相同的原理，并且模仿了该算法。消息分组和填充方式与MD5相同。

SHA-1使用了f_0，f_1，…，f_{79}这样一个逻辑函数序列。每一个f_t（$0 \le t \le 79$）对3个32位双字B，C，D进行操作，产生一个32位双字的输出。f_t（B，C，D）定义如下：

[image]

需要注意的是，在常见的SHA-1实现的程序中，这4个函数经常被定义成如下的形式（C语言）：

[image]

这与上面的f_t（B，C，D）的定义是等价的，将产生相同的散列值。

类似于MD5，SHA-1也使用了一系列的常数K(0)，K(1)，…，K(79)。以十六进制形式表示如下：

[image]

SHA-1产生160位的消息摘要，在对消息进行处理之前，初始散列值H先用5个32位双字初始化，这5个双字以十六进制形式表示如下：

[image]

在解密者尝试分析算法时，可以通过上面的常数K_t及H_t来识别其为SHA-1。

关于SHA-1、SHA-256、SHA-384和SHA-512更加详细的计算过程，请参考FIPS 180-1及FIPS 180-2。最后附上SHA-256、SHA-384和SHA-512初始化数据（十六进制形式）。

SHA-256初始化数据是：

[image]

SHA-384初始化数据是：

[image]

SHA-512初始化数据是：

[image]

2．实例分析

用PEiD的Krypto ANALyzer插件查看光盘中的SHA1KeyGenMe.exe，得知该KeyGenMe中含有SHA-1的压缩常数，猜测可能使用了SHA-1算法（见图6.2）。

[image]

图6.2　用PEiD的插件扫描目标程序的加密算法

用OllyDbg打开光盘中的SHA1KeyGenMe.exe，在命令栏中输入“bpx GetDlgItemTextA”（bpx命令可以将断点下在所有调用GetDlgItemTextA的代码上），按F9键运行KeyGenMe，分别输入Name：pediy和Serial Number：0123456789ABCDEFGHIJ，单击“Check”按钮，程序将中断在第一个GetDlgItemTextA处。

[image]

[image]

[image]

通过上面的代码，可以得知注册码，由此可以做出注册机了。注册机源代码见本书光盘。

6.1.3 小结

以上简要介绍了两种常见的单向散列函数加密算法MD5和SHA-1。除此之外，密码学中的Hash算法还有很多种，如RIPEMD、HAVAL、Tiger等，感兴趣的读者可以参考相关的资料。

需要引起重视的是，随着密码分析技术的发展，现有的散列算法都是不安全的。如SHA-160、MD5、RIPEMD、HAVAL、Tiger在某些条件下能够构造出碰撞。软件保护人员在使用散列算法进行保护时，建议选择SHA-256/384/512，或者使用Whirlpool。

如果在解密时碰到Hash算法，一般只要根据每种Hash算法的特征搞清楚是哪一种Hash算法以及该算法是否变形，继而通过该Hash的源代码即可做出注册机。

6.2 对称加密算法

对称加密算法的加密密钥和解密密钥是完全相同的。其安全性依赖于以下两个因素：第一，加密算法必须是足够强的，仅仅基于密文本身去解密信息在实践中是不可能的，可以抵抗现有的各种密码分析方法的攻击；第二，加密安全性依赖于密钥的秘密性，而不是算法的保密性。

若要采用对称算法检验注册码，正确的使用方法是把用户输入的注册码（或者注册码的一部分，注册码的散列值）作为加密算法或者解密算法的密钥。这样，解密者要想找到一个正确的注册码，只能采用穷举法。为了增大穷举的难度，自然要求注册码有一定的位数。如果在检查注册码时，把用户输入的注册码作为算法的输入或者输出，则无论使用加密算法还是解密算法检查注册码，解密者都可利用调试器在内存中找到所用的密钥，从而可以将算法求逆，写出注册机来。

常见的对称分组加密算法有DES（Data Encryption Standard）、IDEA（International Data Encryption Algorithm）、AES（Advanced Encryption Standard）、BlowFish、Twofish等。本节将以常见的对称加密算法TEA、IDEA、BlowFish、AES及流密码RC4为例，介绍对称算法在软件保护中的应用。感兴趣的读者可以进一步阅读密码学相关书籍，如《对称密码学》（机械工业出版社）来了解更多的关于对称密码的知识。

6.2.1 RC4流密码

RC4于1987年由Ron Rivest设计，当时作为商业秘密并未公开。1994年，其算法描述被匿名发表在Cypherpunks邮件列表中，不久传到sci.crypt新闻组进而在互联网上流传开来。感兴趣的读者可以在这里阅读当时发到sci.crypt的原始帖子：http://groups.google.com/group/sci.crypt/msg/10a300c9d21afca0。时至今日，RC4已经成为最为流行的流密码，如应用于SSL（Secure Sockes Layer）、WEP。随着众多的对其密码分析成果，密码学家认为RC4的安全性不是很强，但在实际应用中还是可以保证一定安全性的。

1．算法原理

RC4生成一种称为密钥流的伪随机流，它同明文通过异或操作相混合以达到加密的目的，解密时，同密文进行异或操作。其密钥流的生成由两部分组成：KSA和PRGA。

（1）KSA（the Key-Scheduling Algorithm）

RC4首先使用密钥调度算法（KSA）来完成对大小为256的字节数组S的初始化及替换。在替换时使用密钥。其密钥的长度一般取5～16个字节，即40～128位，也可以更长，通常不超过256位。首先用0～255初始化数组S，然后使用密钥进行替换。伪代码如下：

[image]

（2）PRGA（the Pseudo-Random Generation Algorithm）

数组S在完成初始化之后，输入密钥便不再被使用。密钥流的生成是从S[0]到S[255]，对每个S[i]，根据当前S的值，将S[i]与S中的另一字节置换。当S[255]完成转换后，操作继续重复执行。伪代码如下：

[image]

得到的子密码k用以和明文进行XOR运算，得到密文，解密过程也完全相同。由于RC4算法加密采用的是XOR，所以，一旦子密钥序列出现了重复，密文就有可能被破解。推荐在使用RC4算法时，必须对加密密钥进行测试，判断其是否为弱密钥。

2．实例分析

RC4算法简单易懂，本书光盘中附有一个使用RC4算法对消息进行加密的例子RC4 Sample，请读者参考详细源代码。下面是相关的汇编代码：

[image]

RC4加密与解密都是调用XOR指令，加密与解密都是调用同一个函数（本例是call 00401070），其密钥也相同。

6.2.2　TEA算法

TEA全称为Tiny Encryption Algorithm，于1994年由英国剑桥大学的David J. Wheeler发明。

1．算法原理

TEA的分组长度为64位，密钥长度为128位。采用Feistel网络。其作者推荐使用32次循环加密，即64轮。其加密过程如下所示：其中K[0]～K[3]为密钥，v[0]～v[1]为待加密的消息：

[image]

其中，delta是由黄金分割点得来的，delta＝（[image]－1）$\times 2^{31}$。解密算法是加密的逆过程，代码如下所示。

[image]

由以上TEA的加密与解密源码可以看出其算法简单易懂，容易实现。但是TEA存在相当大的缺陷，如相关密钥攻击。考虑到TEA的缺陷，密码学家也相继提出了一些改进算法，比如XTEA。

2．实例分析

本书光盘中的TEAKeyGenMe.exe是一个以TEA及MD5算法保护的KeyGenMe，以此为例介绍TEA在软件保护中的应用。

用OllyDbg加载TEAKeyGenMes.exe，按F9键运行输入用户名和假码，用“bpx GetDlgItemTextA”下断可以来到如下代码处：

[image]

[image]

[image]

[image]

[image]

从上面的汇编代码分析可以判断出，这是TEA加密函数，使用用户名的128位散列，对用户输入的注册码的十六进制，共64位，进行加密。

[image]

上面的结果是将TEA的加密结果，共64位，同散列值的高64位进行异或，异或的结果同散列值的低64位进行比较，若二者相同，则注册码正确，否则注册码不正确。

整个注册码验证的逆过程即首先对用户名进行MD5散列，并将其作为TEA的密钥。将散列的高64位同低64位进行异或，将异或的结果存储于缓冲区szBuffer中，然后用TEA对szBuffer进行解密，最终输出解密结果的ASCII字符即为注册码。其详细源代码见本书光盘。

6.2.3　IDEA算法

IDEA（International Data Encryption Algorithm，国际数据加密算法），于1991年由XueJia Lai（来学嘉）和L. Massey提出。

1．算法原理

分组密码IDEA明文和密文的分组长度为64位，密钥长度为128位。该算法的特点是使用了3种不同的代数群上的操作。

（1）子密钥生成

IDEA共使用52个16位的子密钥，其由输入的128位密钥生成，过程如下。

• 首先，输入的128位密钥被分成8个16位的分组，并直接作为前8

个子密钥。

• 然后，128位密钥循环左移25位，生成的新的128位密钥被重新分成8个16位的分组，作为下面8个子密钥。

• 重复上一步，直至52个子密钥全部生成。

（2）IDEA加密算法

IDEA的加密过程由8个相同的加密步骤（称为加密轮函数）和一个输出变换组成。整体结构如图6.3所示。

[image]

图6.3　IDEA加密结构

• [image] 表示按位异或操作；

• [image] 表示定义在模2^{16}（ = mod 65536）的模加法运算，其操作数都可以表示成16位整数；

• [image] 表示定义在模$2^{16}+1$（ = mod 65537）的模乘法运算。

首先，64位明文被分成4个16位分组。每一轮加密需要6个子密钥，最后的输出变换只需要4个子密钥，所以共需要52 = 8 × 6 + 4个子密钥。如图6.3所示，在第一轮加密中，4个16位的子密钥分别通过两个模$2^{16}+1$的乘法运算和两个模2^{16}的加法运算与明文进行混合。结果进一步处理，又用到了两个16位的子密钥以及按位异或操作。第一轮加密的结果，在进行部分交换后，作为第二轮加密的输入。以此再重复进行7轮。在接下来的输出变换（Output Transform）中，使用52个子密钥的最后4个，通过模加与模乘运算与第8轮的结果进行混合，产生最后的密文。

（3）IDEA解密算法

对密文的解密计算过程同对明文的加密过程是一样的，如图6.3所示。解密与加密唯一不同的地方，就是使用不同的子密钥。首先，解密所用的52个子密钥是加密的子密钥的相应于不同操作运算的逆元。其次，解密时子密钥必须以相反的顺序使用。

IDEA中的加法与乘法逆元的规则定义如下：

[image]

其中，模$2^{16}+1$的乘法逆元的计算可以使用欧几里德扩展算法来求，代码如下：

[image]

2．实例分析

本书光盘中的IDEAKeyGenMe是一个使用IDEA算法进行注册验证的KeyGenMe。先用PEiD的Kcrypto ANANLyzer查看发现还有SHA-1算法。用OllyDBG加载，按F9键运行，输入假的用户名与序列号，用“bpx GetDlgItemTextA”下断，单击“Check”按钮中断后可来到如下代码处：

[image]

[image]

[image]

[image]

[image]

通过对上面一段循环代码的分析，可以看出，这是将一个共128位的字（16位）数组wiDeaKey循环左移25位，作为一个新的128位数组送入wSubKey。共生成了56个16位字。由于IDEA算法使用52个16位的子密钥，且同样需要循环左移25位，据此可以大致猜测，此过程为IDEA的子密钥生成函数。在IDEA中，一共执行了8轮加密及1轮输出变换，共需要9个向量：每一轮使用6个密钥，输出变换使用4个，共需要6个向量。紧接着的一段代码是将此56个16位字（IDEA只使用前52个）送入一个二维数组Z[7][10]。Z[0][10]并没有使用，且对于第i行（0 < i < 7），只使用了9列。读者可以对照着图6.3来理解。

[image]

[image]

[image]

[image]

[image]

分析这段代码，结合对IDEA加密算法的原理，可以看出此处就是IDEA的加密函数，即对序列号（十六进制形式）进行加密。在上面提到，szHash的低32位及“C:\”盘卷序列号被存储到了一缓冲区szBuffer。在进行完IDEA加密后，将加密的结果，共64位，同szBuffer进行比较，若相同，则序列号正确，否则注册失败。

其逆过程即为使用用户名160位散列的前128位作为IDEA的密钥，对散列的32位和卷序列号进行IDEA的解密运算，再转化成其ASCII码形式，即为最终的序列号。注意IDEA解密密钥的生成过程，详细源代码见本书光盘。

6.2.4 BlowFish算法

BlowFish算法是一个64位分组及可变密钥长度的分组密码算法，该算法是非专利的。

1．算法原理

BlowFish算法基于Feistel网络（替换/置换网络的典型代表），加密函数迭代执行16轮。分组长度为64位（bit），密钥长度可以从32位到448位。算法由两个部分组成：密钥扩展部分和数据加密部分。密钥扩展部分将最长为448位的密钥转化成共4168字节长度的子密钥数组。其中，数据加密由一个16轮的Feistel网络来完成。每一轮由一个密钥相关

置换和一个密钥与数据相关的替换组成。

（1）子密钥

BlowFish使用大量的子密钥。这些密钥必须在进行加密前预计算产生。

• P数组由18个32位字的子密钥组成：P1,P2,…,P18。

• 4个8×32的包含总共1024个32位字的S-box：

[image]

子密钥扩展算法如下：

① 按顺序使用常数π的小数部分初始化P数组和S-box。

比如：

[image]

② 对P数组和密钥进行逐位异或，需要时重用密钥。

③ 使用当前的P数组和S-box对全0的64位分组使用BlowFish算法进行加密，用输出替代P1、P2。

④ 使用当前的P和S对第③步的输出进行加密，并用输出替代P3、P4。

⑤ 继续上面的过程，直到按顺序替代所有的P数组和S-box中的元素。

（2）加密

BlowFish是由16轮的Feistel网络组成的。输入是一个64位的数据元素x，将x分成两个32位部分：xL,xR。加密算法的伪C代码如下：

[image]

其中“^”表示异或运算，函数F的输入是一个32位双字，共4个字节，分别作为4个S-box的索引，取出相应的S-box值，然后进行模2^{32}加运算。用等式可以描述如下：

[image]

解密与加密完全相同，只不过P1,P2,…,P18以相反的顺序使用。

2．实例分析

先用PEiD的Krypto ANALyzer插件查看BlowFishKGM.exe，可以识别出BlowFish的P数组和S-box，猜测该KeyGenMe使用了BlowFish算法。

用OllyDbg打开本书光盘中的BlowFishKGM.exe查找参考字符串，可以找到“Success!”与“Wrong Serial!”，双击“Success!”，可以来到反汇编窗口。向上可以找到注册码判断的地方，从00401123处开始，在此下断点。按F9键运行，输入Serial Number：

123456789ABCDEFFEDCBA987654321

可以断在00401123处。代码如下：

[image]

[image]

[image]

[image]

[image]

[image]

紧接着，KeyGenMe得到C盘的硬盘序列号，与上面得到的异或的结果进行比较，如果相等，则表示注册码正确，否则注册失败。

整个序列号验证算法清楚了。即用输入的注册码的前16个字节的十六进制形式作为BlowFish算法的密钥，对其后的16个字节的十六进制码进行解密，将解密得到的两个双字进行异或，其结果需和硬盘序列号相等。其逆算法为，可随机生成12个字节，其前8个字节作为BlowFish的密钥，剩下的4个字节作为一个双字与硬盘序列号进行异或得到另外一个双字，然后对这两个双字（共64位）进行BlowFish加密运算，得到的64位密文即为注册码的后8个字节（十六进制形式），然后调用sprintf函数将8个字节的密钥及加密后的8个字节的密文进行输出，即是最后的注册码了。更加详细的过程请参考光盘中提供的注册码源代码。

6.2.5 AES算法

AES即Advanced Encryption Standard，高级加密标准。是NIST（National Institute of Standards Technology）于1997年开始向世界范围内征集的加密算法，用于替代DES成为新一代的加密标准。1997年9月NIST发布了AES需要符合的标准，其中要求AES具有128比特的分组长度，并支持128、192和256比特的密钥长度，而且要求AES能在全世界范围内免费得到。AES的评选工作一共进行了3轮。第一次共有15个算法入选，分别为CAST-256，CRYPTON，DEAL，DFC，E2，FROG，HPC，LOKI97，MAGENTA，MARS，RC6，RIJNDAEL，SAFER+，SERPENT，TWOFISH。在第二次的公开评选后，NIST宣布共有5个算法进入最后决赛，分别是MARS，RC6，Rijndael，Serpent，Twofish。最终于2000年10月，NIST宣布Rijndael由于在各方面的表现都十分优秀，当选为AES。

Rijndael由两位国际著名的比利时密码学家Joan Daemen和Vincent Rijmen设计，读作“Rain Doll”。实际上，Rijndael算法本身和AES的唯一区别在于各自所支持的分组长度和密码密钥长度的范围不同。Rijndael是具有可变分组长度和可变密钥长度的分组密码，其分组长度和密钥长度均可独立地设定为32比特的任意倍数，最小值为128比特，最大值为256比特。而AES将分组长度固定为128位，而且仅支持128、192和256位的密钥长度，分别称作AES-128、AES-192、AES-256。在本书中提到的AES，如无特别说明，专指FIPS-197中规定的AES算法。

1．基本术语

（1）字节

AES算法的基本处理单元叫做字节，它由8比特序列组成，被看作为

一个整体。在AES中，这些字节以有限域（Finite Field）上的多项式（polynomial）来表示：

[image]

其中，b_i（$0 \le i \le 7$）分别代表了一个字节的8个比特位。如字节{01100011}，即0x63，代表了相应的有限域元素$x^6 + x^5 + x + 1$。

（2）状态（State）

AES的所有操作都是在一个称作状态（State）的二维字节数组上进行的。状态由4行字节组成，每行包括Nb个字节，Nb为分组长度除以32的值。用s来表示状态，状态数组中的每个字节有两个坐标，行号r的范围为$0 \le r < 4$，列号c的范围为$0 \le c < Nb$。这样就可以用$S_{r,c}$或者s[r,c]来引用状态中的每个字节。在AES算法的加密和解密过程的开始，输入字节数组in_0，in_1，…，in_{15}复制到状态数组中，然后对状态数组中的元素进行加解密操作，最后将结果复制到输出字节数组out_0，out_1，…，out_{15}，如图6.4所示：

[image]

图6.4　状态数组输入、输出

在实际的软件实现中，将每一列的4个字节作为一个32位字。如一个长度为128位的分组数据块在内存中的形式如下：

[image]

那么此状态数组为：

[image]

2．数学背景

（1）加法

有限域上的两个元素的加法运算是通过对相应的多项式中的相同次幂的系数进行相加来实现的。其加法是模2加运算，也就是异或操作，用符号“⊕”来表示。相应的减法运算和加法是完全相同的。

例如，下面的表达式是相同的：

[image]

（2）乘法

如果一个多项式的因子为1和它本身，那么称这个多项式是不可约的，此多项式为不可约多项式。以多项式形式表示的GF（2^8）（注：GF表示Galois Field，伽罗瓦域，Galois是第一位研究有限域的数学家）上的乘法运算（以·表示），对应于多项式相乘然后再模一个8次的不可约多项式（关于有限域更加详细的论述，请参考《Introduction to Finite Fields and Their Applications》一书）。对于AES算法，这个不可约多项式为：

[image]

或者以十六进制表示为｛01｝｛1b｝，即0x11B。

例如，｛57｝·｛83｝＝｛c1｝，因为

[image]

通过m（x）模约简运算，可以保证结果为一个次数小于8的二进制多项式，并且可以用字节来表示。同时，AES还涉及了有限域上求乘法逆元的运算，读者可以参考相关资料。在后面的章节中，还会涉及模p的乘法逆元计算问题，将在后面讲述。

（3）与x相乘

将上面定义的式（6-1）与多项式x相乘，其结果为：

[image]

然后将其进行模m（x）约简，即如果$b_7 = 0$，结果就是最简形式。若$b_7 = 1$，则减去多项式m（x）（减法运算即上面讲到的异或运算）。这种乘法运算在AES中称作xtime()。如果乘以x的高阶次幂，只要重复进行xtime()运算，并将每个中间结果进行相加即可。

例如：｛57｝·｛13｝＝｛fe｝，因为：

[image]

所以

[image]

3．算法描述

对于AES算法来讲，输入分组、输出分组及状态数组的长度都是128比特，即Nb＝4。密钥K的长度为128、192或者256比特，用Nk＝4、6或者8来表示。加密或者解密函数所执行的轮数取决于密钥的长度。轮数用Nr来表示，则当Nk＝4时，Nr＝10；Nk＝6时，Nr＝12；Nk＝10时，Nr＝14，如表6-1所示。

表**6-1　AES**算法描述

[image]

（1）加密过程

先将输入复制到状态数组。在进行一个初始轮密钥加（Round Key Addition）操作之后，执行Nr次轮函数（Round Function）对状态数组进行变换，其中最后一轮不同于前Nr-1轮。最终的状态数组复制到输出即为最后的密文。

轮函数由4部分组成，分别是SubBytes()、ShiftRows()、MixColumns()和AddRoundKey()。其加密过程用伪代码表示如下：

[image]

（2）SubBytes()

SubBytes，字节代换，实际上就是一个简单的查表操作。AES定义了一个16×16个字节的S盒（S-box），如表6-2所示。以状态数组中的每个字节元素的高4位作为行标，低4位作为列标，取出相应的元素作为SubBytes操作的结果。比如十六进制值{C5}，高4位为C，低4位为5，取S盒中的行标为C列标为4的元素为十六进制{A6}，则{C5}将被替换为{A6}。

表6-2　S-box（十六进制）

[image]

按照此种操作，将状态数组中的所有元素都替换为S盒中的值。关于S-box的详细设计原理，请参考相关资料。

（3）ShiftRows

ShiftRows操作规则是状态数组的第一行保持不变，第二行循环左移一个字节，第三行循环左移两个字节，第四行循环左移三个字节。比如状态：

[image]

经过ShiftRows操作后的结果为：

[image]

（4）MixColumns

MixColumns操作是以列为单位的，把状态中的每一列看作一个系数在$GF(2^8)$上的四项多项式，然后乘以一个固定的多项式a(x)，再模（x^4+1）。a(x)的定义如下：

[image]

记状态S经过MixColumns操作后为S'，则MixColumns可以看作是一个矩阵乘法。

[image]

其中$0 \le c < Nb$，上面的矩阵乘法即

[image]

（5）AddRoundKey()

AddRoundKey操作是将状态中的元素同轮密钥通过简单的异或运算相加。轮密钥是由用户输入的密钥通过密钥扩展过程而生成的，同样可以看作一个状态数组。

（6）密钥扩展（Key Expansion）

密钥扩展算法通过对用户输入的128、192或者256位的密钥进行处理，共生成Nb（Nr+1）个32位双字，为加解密算法的轮函数提供轮密

钥。密钥扩展算法伪代码如下：

[image]

其中，SubWord函数以一个4个字节的双字作为输入，然后对每个字节利用S盒进行替换作为输出。函数RotWord以双字$[a_0,a_1,a_2,a_3]$作为输入，做循环左移操作，输出为$[a_1,a_2,a_3,a_0]$。轮常量数组Rcon中的每一个元素Rcon[i]为一个32位双字，且低24位恒为0。高8位，即一个字节按如下规则定义：Rcon[1] = 1，Rcon[i] = 2*Rcon[i-1]，乘法定义于$GF(2^8)$上。以10轮为例，Rcon的值为（十六进制）：

[image]

（7）解密过程

加密算法的逆过程即为解密算法。因此解密算法的轮函数由4个部分组成，分别为InvShiftRows()、InvSubBytes()、InvMixColumns()和AddRoundKey()。

InvShiftRows是ShiftRows的逆过程，即状态中的后三行执行相应的右移操作，如第二行循环右移一个字节。

InvSubBytes是SubBytes的逆过程。AES同时也定义了一个逆S盒（Inverse S-box）。同SubBytes一样，InvSubBytes也只是简单的查表操作。

InvMixColumns与MixColumns的原理是相同的，不同的是使用了不同的多项式。

[image]

InvMixColumns使用的系数矩阵为[image]它同MixColumns中使用的矩阵互为逆矩阵。

AddRoundKey的逆过程就是它本身，因为异或操作是其本身的逆。

4．在32位处理器上的实现

在加解密及逆向工程中，常见的AES算法的实现与上面所讲述的过程是不同的。实际的软件实现采用了以空间换时间的方法，将轮函数的几个步骤合并为一组简单的查表操作。

假设轮函数的输入用a表示，SubBytes的输出用b表示，则

[image]

又设ShiftRows的输出用c表示，MixColumns的输出用d表示：

[image]

[image]

式（6-4）中下标的加法必须是模Nb的。式（6-3）~式（6-5）可以合并为：

[image]

定义4个表：T_0，T_1，T_2，T_3

[image]

每个T表都有256个4字节的32位双字，从而需要4KB的存储空间。使用这些表，可以将式（6-6）改写成：

[image]

同时，AddRoundKey可以通过在每一列上执行一个额外的32位异或运算来实现，所以使用该4KB的表，对每一轮的每一列只需要4次查表和4次异或运算。而且这4个表只需要一个即可，其他三个可以通过循环移位得到。由于最后一轮没有MixColumns操作，所以最后一轮仍然需要使用常规的方法。

下面以通过查上面所讲到的4KB的表实现AES，来讲解AES在软件保护中的应用。

5．实例分析

先用PEiD的Krypto ANALyzer查看光盘中的AESKeyGenMe.exe，结果如图6.5所示。

[image]

图6.5　显示PEiD分析结果

这表示该KeyGenMe中有用于MD5算法的64个常量元素的T表及AES的S盒和逆S盒，可以猜测使用了MD5和AES两种算法。下面具体来分析该KeyGenMe。首先用OllyDbg加载，按F9键运行，输入假的用户名和序列号，用“bpx GetDlgItemTextA”下断，可以来到如下代码处：

[image]

[image]

上面的这段代码是对用户名使用MD5算法进行散列，根据前面对MD5的讲解，很容易判断出来。此处将产生128位的散列。

[image]

00401264处的“lea eax,[esp + 2C]”，[esp + 2C]指向地址0012F4F4，其数据如下：

[image]

共128位，这是AES的密钥。可以看出这是AES-128。aes_init函数共有5个参数：aes_init(aes* a,int mode,int nk,char* key,char* iv)

• a是一个AES结构，定义了AES的内部参数。

• mode是AES的工作模式，关于对称算法的工作模式请参考相关资料。由于此处只对一个128位分组进行处理，所以采用ECB模式。

• nk为密钥的长度，这为16个字节，即128位。

• key为指向密钥数组的地址。

• iv是初始化向量，用于CBC等模式，ECB不需要，所以设为NULL。

跟进401EC0，代码如下：

[image]

[image]

从前面介绍的AES密钥扩展算法可以知道，子密钥数组dw[i], i > 4，依赖于dw[i-1]和dw[i-Nk]。此例中Nk = 4，即密钥长度为4个32位双字。所以子密钥数组只与dw[i-1]和dw[i-4]有关。对dw数组中下标为4的倍数的元素，将采用如下的方法生成子密钥：

[image]

下面是AES加密子密钥生成的主要代码：

[image]

[image]

[image]

在本例的密钥扩展函数中，同时也生成了用于解密的子密钥，请读者自行分析。

[image]

跟进4023A0处，可以发现程序先判断是何种工作模式。本例中为ECB模式，所以直接调用aes_ecb_encrypt函数。上面提到过通常AES的加解密过程的实现是通过查4个表T_0、T_1、T_2、T_3来实现的，本例中即是这种方法。

首先是进行一次AddRoundKey：

[image]

接着执行轮函数，主要代码如下：

[image]

[image]

[image]

上面共执行了9轮一样的轮函数，在前面讲过，AES的最后一轮不同于前面的Nr-1轮。对于加密过程少了一个MixColumns操作。最后一轮的主要代码如下：

[image]

[image]

[image]

上面的这段代码是产生一个32位双字的输出y[0]，其他的三个输出用同样的方法，请读者自行分析。将y数组复制到输出缓冲区即为最终的密文。

然后将密文同用户名的128位MD5散列值相比较，若相同，则注册成功，否则失败。写注册机只需要对用户名的128位MD5散列值进行AES解密，即可得到序列号。详细的源代码请参考光盘。

6.2.6 对称加密算法小结

除了前面介绍的几种分组密码外，还有许多的分组密码没有介绍，如经典的DES、有趣的Twofish、安全性极高的Safer＋，以及NESSIE里最新提交的MISTY1、Camellia等。另外，关于分组密码的工作模式也没有过多的讨论，读者可以通过阅读密码学专著来进一步了解。

如果软件中使用了对称加密算法，那么一般来说，只要知道了算法的类型及密钥，那么就可以做出注册机来。可以用如下方法识别软件中所使用的对称加密算法。

（1）使用PEiD的Krypto ANALyzer（Kanal）插件进行识别，一般的对称加密算法都可以识别出来，但也有例外（如IDEA）。需要注意的是，不能依赖于工具。使用工具只是一种辅助，还需要进一步的跟踪以确定到底是何种算法。

（2）通过每种加密算法的独特的加解密处理过程，如是否为Feistel网络，加密轮数，密钥长度，子密钥生成过程，S-box的值等一系列信息来区分和确定软件中所使用的算法。

（3）为了进一步确定是否为某种对称加密算法，以及此种算法采用何种工作模式（ECB，CBC，CFB，CTR等），往往还需要自己写一个此种算法的加解密程序来和软件中的算法进行对比检验。

6.3 公开密钥加密算法

在上一节中介绍的对称加密算法，其加密与解密都使用同一个密钥，一旦知道了密钥，那么保护就失败了，这也是其缺点。基于这种考虑，国际著名密码学家Diffie. W和Hellman. M. E于1976年，在其发表的文章“New Directions in Cryptography（密码学的新方向）”中提出了公开密钥（Public Key）加密算法的概念。公钥算法加密与解密使用不同的密钥，加密所使用的叫做公钥（Public Key），而解密所使用的叫做私钥（Private Key）。故而，公钥加密算法又称为非对称加密算法（Asymmetric Key Cryptography）。任何人都可以使用密钥分配者所

分发的公钥对信息进行加密，而只有私钥的所有者才可以解密。

公开密钥的设计都是基于NP完全问题（关于NP问题的详细介绍，请参考数论及密码学相关方面的资料，几乎任何一本讲解密码学的书都会涉及）。如1978年，麻省理工学院的三位教授Rivest、Shamir及Adleman提出的一种基于因子分解问题的公钥系统，它就是现在应用十分广泛的RSA公钥算法。后来所出现的背包公钥密码系统（Knapsack）、Elgamal公钥密码、ECC等无一不是基于NP问题所设计的。

如果软件作者在生成注册码时采用解密算法（私钥），而在软件中检查注册码时使用加密算法（公钥），即使解密者能够用调试器在自己的机器上对软件进行跟踪分析从而找到公钥，他也不一定能够计算出私钥，自然也就无法得到正确的注册码，更无法写出注册机来。

6.3.1　RSA算法

RSA是第一个既能用于数据加密也能用于数字签名的算法，易于理解和操作，应用十分广泛。算法的名字以发明者的名字命名：Ron Rivest、Adi Shamir和Leonard Adleman。密码分析者既不能证明也不能否定RSA的安全性，但这恰恰说明该算法有一定的可信度。

1．算法原理

① 选取两个大素数：p和q，为了获得最大程度的安全性，两数的长度一样。（注：以下所涉及的数论知识不做过多说明，感兴趣的读者请进一步参阅相关书籍。）

② 计算$n = p \times q$，n称为模。

③ 计算欧拉（Euler）函数：$\varphi(n) = (p-1) \times (q-1)$。

④ 选取加密密钥e，其与φ(n)互素。如果选择合适的e值，RSA加解密的速度将快得多，常用的e为3、17和65537（$2^{16}+1$）。

⑤ 使用扩展欧几里德算法（Extended Euclid）求出e模φ(n)的逆元d，即

[image]

⑥ 公钥为e和n，私钥为d，p和q可以丢弃，但是必须保密。

⑦ 加密消息m时，将其看成一个大整数，把它分成比n小的数据分组，按下面的式子进行加密：

[image]

⑧ 对密文c解密时，取每一个加密后的分组c_i并计算：

[image]

RSA加解密总结见表6-3。

表6-3　RSA加解密

[image]

RSA的安全性依赖于大整数因子分解，但是否等同于大整数因子分解一直未能得到数学上的证明，也就是说，没有证明要解密RSA就一定要进行因子分解。目前，RSA的一些变形算法已被证明等价于大整数因子分解问题，如Rabin公开密钥系统。但是目前来看，攻击RSA算法最有效的方法便是分解模n。随着分解大整数方法的改进、计算机速度的提高以及计算机网络的发展（可以使用互联网上成千上万的计算资源同时进行因子分解），为了保证RSA系统的安全性，其密钥的位数一直在增加。目前，一般认为RSA需要1024位或更长的模数才有安全保障。常见的因子分解算法有试除法（Trial Division）、Pollard-ρ因子分解算法、Pollard p-1因子分解算法、椭圆曲线因子分解算法（Elliptic Curve Factoring Algorithm）、随机平方因子分解算法（Random Square Factoring Algorithm）、连分式因子分解算法（Continued Factoring Algorithm）、二次筛法（Quadratic Sieving）、数域筛法（Number Field Sieving）（包括一般（广义）数域筛法（GNFS）和特殊数域筛法（SNFS）），本书不作过多介绍，详细请参考有关资料。

2．RSA计算

RSA涉及的各公式的计算请参考数论等相关资料。在本书光盘里直接提供了相关计算工具。

• Gcd.exe：求最大公因子；

• MulInv.exe：求模逆元，形式为$ed \equiv 1 \bmod \varphi(n)$，其中$\varphi(n) = (p-1)(q-1)$；

• Powmod.exe：计算$m \equiv c^d \bmod n$；

• CE.EXE：计算d，用法：CE < p > < q > < e >；

• Factor：大数计算器，可进行因式分解；

• RSATool：一款非常强大的RSA辅助工具，具有图形界面，包含上述工具功能（注意，输入十六进制时，字母一定要以大写形式输入）。

• Bigcalc：大数计算器。

下面举一个例子。

设p = 37，q = 41（十进制），那么：

[image]

选取e = 17，则$d = 17^{-1} \bmod 1440 = 593$，公开e和n，将d保密，丢弃p和q（但也必须保密）。加密消息：

[image]

首先将其分成小的分组，在此例中，按三位数字一组就可以进行加密。

[image]

$m_3 = 007$（不足在左边填充0）

加密：

[image]

密文如下：

[image]

解密消息时需要私钥593进行相同的指数运算。例如：

[image]

3．RSA算法在加密上的应用

大多数共享软件的注册码计算设计得都不是很好，比较容易被解密做出注册机来。如果采用公钥算法作为注册保护机制，如RSA，可以参考如下思路。

① 要求其模数n有一定的长度（至少为512位，推荐取1024位或更长），以防止在较短的时间内被因式分解，从而使算法被攻破。但注册码的长度也因此变长了，可能给用户带来不方便，可以采取以License file（授权文件）的形式分发给注册用户。

② 随机生成密钥对时，要采用尽可能好的随机数生成算法，以免被轻而易举地猜到公钥或私钥。

③ 也可以在注册机中用公钥e对用户名进行加密得到注册码，在软件中对用户输入的注册码用私钥d进行解密得到用户名。此时公钥e就不能取常用的3、65537等值，否则一旦被猜出或计算出e，也可以做出注册机。

④ 这种方法只是为了防止被解密者写出注册机，无法防止通过修改程序中跳转指令的方法来破解软件。为了防止别人修改程序文件，可以用注册码中的一部分来加密程序代码或数据。

下面举一个例子讲解RSA应用。该例采用大数运算库Miracl来实现RSA，该库提供C与C＋＋两种接口。读者可参考其说明文档将其安装好。

（1）随机生成密钥对

可以自己编程随机搜索大素数，请参阅相关的资料。此处由于是举例，采用RSATool工具生成128位RSA的参数：

[image]

（2）在软件中判断注册码

在软件中用公钥e对输入的注册码进行加密来得到密文，并与用户名比较。若相同，则认为是注册成功，否则注册失败。

[image]

（3）制作注册机

思路是：将用户名c用私钥d解密，得到的数据为注册码。代码如下：

[image]

由于m往往包含不可显示字符，必须将其转换成十六进制字符串，或将其编码变成可显示字符，比如采用Base64编码等。

4．攻击RSA保护

采用RSA保护时，模数n的位数不能太少。实际中有些共享软件虽然采用了RSA，但n太短，这样就失去了用RSA保护的意义。攻击RSA保护的软件，一般是通过跟踪分析得到n，再将n因子分解，从而求出私钥d，进而做出注册机。

以光盘中的RSAKeyGenMe.exe为例。用OllyDbg运行此KeyGenMe，输入假的姓名与序列号，用“bpx GetDlgItemTextA”设断，来到如下代码处：

[image]

[image]

从上面的代码中获得了两个重要的参数模n和公钥e，其值分别为：

[image]

因为n并不是很大，直接将其因式分解得到p和q。因式分解可以用RSATool，Factor或PPSIQS等工具。RSATool工具是图形界面，操作比较方便。选择进制（Number Base）为16，将模数80C07AFC9D25404D6555B9ACF3567CF1填入“modulus(N)”中，单击Factor N按钮进行因式分解。128位的大数在不到一分钟的时间内就被分解了：

[image]

即：

[image]

知道p和q，便能计算出φ（n）=（p-1）（q-1），既而利用欧几里德扩展算法很容易求出d，利用RSATool，输入e后，单击“Calc.D”按钮便可计算出d：

[image]

至此，这个RSA-128被破译。可以直接写出注册机了，源码见光盘。

这里讲解一下如何用工具解密单个数据。设输入的用户名m为pediy，其ASCII码的十六进制值为：

[image]

生成注册码c的加密算法为：

[image]

即

[image]

这时就需要用到Bigcalc这个大数计算器了。先设置进制（Base）为16，设：

[image]

以输入X为例，在Input&Output窗口中，输入7065646979，接着单击“Store”按钮，然后单击“X”按钮，这时7065646979已经存储到变量X中去了。

利用相同的办法，存储Y和Z，最后单击“X^Y%Z”按钮计算$c \equiv m^d \bmod n$，这时同样在Input&Output窗口中得到结果：404E85B5FEF4AE26FC2229D028BE01ADh。

此例中的RSA模n的长度为128位，但是如果软件的模数n为512或更高，除非是特殊的一些n，可以用特殊的因子分解算法来分解，但需要太多的时间，如果不是出于科学研究因子分解算法的目的，那么就没有必要花费如此的精力。可以采用另外一种技术：替换n，即首先通过编程或利用RSATool之类的工具生成与目标软件中的n相同位数长度的n（此时，其私钥d、p和q已知），然后利用逆向技术，用此n去替换软件中的n，然后用自己的d来做出注册机。

6.3.2 ElGamal公钥算法

1985年，T. ELGAMAL教授在他的一篇论文“A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms（一种基于离散对数的公钥加密系统和签名体系）”中提出了ElGamal公钥算法。其安全依赖于在有限域上计算离散对数的困难性。

1．算法原理

密钥对产生办法。首先生成一个大素数p，模p的乘法群[image]上的一个生成元g和一个小于p的大数x（$1 \le x \le p-2$），然后计算：

[image]

公钥为y、g和p，私钥是x。其中g和p可以由一组用户共用。

（1）ElGamal签名

对消息M签名时，生成一个随机数k，$1 \le k \le p-2$，且gcd(k,p - 1）=1，即k与p - 1互素，计算：

[image]

再用扩展欧几里德算法对下面方程求解b：

[image]

当k与p - 1互素时，b有一个解。签名即为（a,b）。随机数k必须丢弃。

验证签名时，需要满足下式：

[image]

同时要检验是否满足$1 \le a \le p$，否则签名容易伪造。

（2）ElGamal加密算法

被加密信息为M，首先选择一个随机数k，且k与p - 1互素，计算：

[image]

其中（a,b）为密文，是明文的两倍长。解密时计算：

[image]

由此也可以看出，ElGamal有一个不足之处就是密文的长度是明文的两倍。而另外一种签名算法，Schnorr签名系统的密文比较短，这由其系统内的单向散列函数h来决定，感兴趣的读者可以参考相关资料。

2．ElGamal算法在加密上的应用

ElGamal提供了签名和加密两种算法。本例利用ElGamal签名算法生成共享软件注册码。思路如下所述。

（1）随机生成密钥对

此处采用RSATool工具生成128位大素数p（实际操作中建议p选取512位以上）和随机数g和x。各数据如下：

大素数：[image]

随机数：[image]

随机数：[image]

计算： [image]

公钥为y、g和p：私钥为x，且必须保密。

（2）在软件中判断注册码

将输入的注册码分成两部分a（签名）和b（签名），再用MD5算法取得用户名的散列值，最后用算式$y^a a^b \equiv g^M \bmod p$验证。若相同，则认为是注册成功，否则注册失败。

用大数运算库MIRACL实现的主代码如下：

[image]

注册机编写过程就是用私钥x生成对数a和b，并将a和b连接起来即可。

3．攻击ElGamal算法保护

ElGamal在签名验证过程中用到了参数y、g和p。解密的思路就是根据$y \equiv g^x \bmod p$，求离散对数获得x的值，即可攻破。

以ElgamalKGM.exe为例演示其过程。用OllyDbg载入，输入假的用户名和序列号，用“bpx GetDlgItemTextA”设断，按F9键运行，可来到如下代码处：

[image]

[image]

以上代码的主要功能是将注册码以“-”为分隔，将其分成两部分，分

别复制到两个缓冲区当中，作为下面将要用到的待验证的签名a,b。

[image]

通过对前面介绍MD5章节的学习，不难分析出上面的代码是标准的MD5，即对字符串“pediy + 用户名”进行散列，得到128位的散列值。

[image]

上面的这些代码主要实现了初始化一些大数变量以及将MD5散列值转化为大数变量的功能。程序从这里开始调用miracl库函数对大数进行处理。如果要看懂这些代码，需要对miracl库进一步熟悉。在稍后的章节中，将介绍如何识别miracl中的函数。

[image]

上面这段代码初始化ElGamal公钥系统的参数p、g和y，以及将要被验证的签名a,b。可以猜想下面即将进行ElGamal的签名验证。

[image]

上面这段代码首先两次调用了powmod函数，分别计算：

[image]

接着调用了miracl中的mad函数，其详细说明请读者参阅miracl手册。这里用于实现计算：

[image]

即计算：

[image]

[image]

可以看出程序紧接着计算下式的值：

[image]

即result3和result4分别是签名验证公式的左右两边的值。然后对result3和result4进行比较，若相等，则签名是有效的，注册成功；否则，签名无效，注册失败。

从以上的分析可以看出，这是标准的ElGamal签名验证算法。经过跟踪分析得到的参数如下：

[image]

现在，最关键的是通过上面的信息来想办法得到其私钥x，已知x满足下面的式子：

[image]

求x这是一个在有限域上的离散对数问题，而这也是数学中的一个NP完全问题。如果p的位数足够大，那么在现有的计算条件及算法水平上，是无法得到x的。但是，随着密码学家和数学工作者的不懈努力，在求有限域的离散对数问题上提出了许多算法。现在，比较常见的求离散对数的算法有Baby-Step Giant-Step Algorithm（大步小步法）、Pollard-ρ Algorithm、Pohlig-Hellman Algorithm、Index-Calculus Algorithm，虽然这些算法仍不能完全解决有限域上的离散对数问题，但是这些算法相当有意思，需要相当的数学功底才能读懂并且通过程序来实现，感兴趣的读者可以参阅相关资料。

求离散此例中的p只有56位，可以通过本书光盘中提供的DLPTool来求出x，此工具只实现了Pollard-ρ算法。利用此工具可以得到：

[image]

在做注册机的时候，需要计算ElGamal的签名（a,b）。随机生成一个k，利用下式计算a：

[image]

但是需要注意的是，k必须与p - 1互素，这样才能保证能得到唯一的b。此例中：p = 0xCE892335578D3F，p - 1 = 0xCE892335578D3E，因子分解后可以得到其两个素数因子f_1 = 2，f_2 = 0x6744919AABC69F，那么在生成k的时候，只要k不包含这两个素因子就可以了，即gcd（k,p - 1） = = 1。

因为b（签名）满足：M ≡ （xa + kb）mod（p - 1）

可以推算出：

[image]

这里k^{-1}是k模p - 1的乘法逆元，可以用扩展欧几里德算法（Extended Eculidean Algorithm）求得。式（6-7）也可以如下表示：

[image]

也就是说，b是（m - xa）k^{-1} mod p - 1的剩余系中的元素。将a和b通过“-”连接起来，便得到了最后的注册码，注册机源代码见光盘。

ElGamal算法的一些注意事项：

（1）如果m - xa出现负值，即最后得到的签名b为负数，一般来讲此时需要将b不断地加上p - 1，直到出现一个小于p - 1的非负值时为止，此时这个非负值便是要求的签名b。在本例中，m为MD5散列值，共128位，转化为128位的大数，私钥x和签名a都是小于等于56位的，其乘积小于等于112位，因此m - xa将恒得到正数，无须考虑签名出现负数的情况。

（2）签名时随机生成的k必须要同p - 1互素，g必须为乘法群[image]的一个生成元。

（3）当签名出现b = 0的时候，一定要重新对消息进行签名，如若

不然，则很容易求出私钥x，这一点要引起重视。

（4）共享软件作者在给用户分发注册码时，一定不要用同一个k对不同的用户名进行签名，并将签名分发给用户，而要采用不同的k，每一个用户对应一个k，或者每个用户对应一对（x,k），并且要定期更换模p，这样才能达到最大程度上的安全。

如果没有采用第3个建议，而采用同一个k和私钥x对不同的用户进行签名，那么对于ElGamal存在这样一种攻击。讲解如下：

用户A和B，使用相同的k对其进行签名，得到的签名分别为（a_1,b_1），（a_2,b_2），其用户名的散列值分别为M_1,M_2，$m_1 \equiv M_1 \bmod p-1$，$m_2 \equiv M_2 \bmod p-1$。令：

[image]

有下式：

[image]

根据模算术运算的性质，式（6-9）-式（6-8）（或者式（6-8）-式（6-9））得：

[image]

设$d=\gcd(b_2-b_1,p-1)$，可以证明$d \mid (m_2-m_1)$。令

[image]

则$m' \equiv kb' \bmod p'$，所以$k \equiv (b')^{-1}m' \bmod p'$，进而可以利用式（6-8）或式（6-9）求出私钥x。

使用miracl库，很容易实现上述的攻击方法。可见，一定不要用同样的k和私钥x对不同的用户名进行签名，只要解密者想办法得到两个或两个以上正确的注册信息，那么就不需要通过求离散对数，也可以求出私钥x，进而做出注册机。这种保护就算是失败的。生成随机数k的算法在注册机源码中已经给出，读者可以参考。

另外，往往在实际使用中，模数p的位数一般都比较大，通过求离散对数解出私钥x一般不可行。此时，可以采取类似于上节介绍的RSA中的“patch n（替换n）”的方法来达到破解的目的，即生成一个同软件中的p相同位数的ElGamal系统参数，替换掉目标软件中的参数，进而做出注册机。

6.3.3　DSA数字签名算法

美国国家标准与技术局（NIST）于1991年，在借鉴了ElGamal及Schnorr签名算法的基础上，公布了数字签名标准（Digital Signature Standard）。该标准采用的算法为DSA（Digital Signature Algorithm）。DSA在公布之后立即产生了巨大的反响，有赞成的也有反对的。因DSA出自NSA（美国国家安全局）之手，由NIST采用并公布，工业界没有任何插手余地，再加上公布之初，DSA仍然存在一些考虑不周到的地方，故而几经修改，直到今天的版本FIPS 186-2（Federal Information Processing Standards，联邦信息处理标准）。目前，对于

DSA的攻击依然在继续，但是仍然没有充分的证据证明其安全性存在很大的弱点。DSA的应用也越来越广泛。

算法原理

DSA使用如下的一些参数：

p：L位长的素数。L是64的倍数，范围从512到1024，$2^{L-1} < p < 2^L$；

q：p - 1的素因子，取值范围为$2^{159} < q < 2^{160}$；

g：$g = h^{(p-1)/q} \bmod p$，其中h满足$h < p - 1$，且$h^{(p-1)/q} \bmod p > 1$；

x：$0 < x < q$，其中x为私钥；

y：$y = g^x \bmod p$，其中（p,q,g,y）为公钥；

k：随机或伪随机数，$0 < k < q$。

整数p、q和g可以公开，并且可由一组用户共享，每个用户分别具有各自的私钥x和公钥y。为了保证最大限度的安全，x和y需要在一段时间内进行更新。参数x和k仅用于生成签名，必须保密。对每个不同的签名，k必须不同（这与上节提到的ElGamal中的k也必须不同的原理相同）。

签名及验证协议如下。

输入：待签名的消息M，公钥p,g,q，私钥x，随机数k

输出：签名r，s

算法：

[image]

k^{-1}是k模q的乘法逆元，SHA-1(M)是指使用SHA-1散列算法对消息进行散列，进而产生160位的输出。

[image]注意：如果在签名的过程中，k,r,s三者有一个为0，那么必须重新生成随机数k，重新进行签名！否则解密者将很容易求出用户的私钥x。

DSA签名验证算法：

输入：待验证的消息M'，公钥p,g,q，公钥y，签名r'，s'

输出：若签名正确则返回TRUE，否则返回FALSE

算法：

（a）如果r'∈（0,q）并且s'∈（0,q），则进行签名验证，否则签名无效，返回FALSE。

（b）在满足（a）的前提下，进行如下计算：

[image]

若v = r'，则签名验证成功。

在使用DSA数字签名系统时，程序员可以自己编程实现生成系统参数，也可以使用DSA Tool来生成。

本书光盘中附有一例DSA Sample，笔者简单地实现了DSA的签名及验证过程。请读者参考光盘中的源码。

DSA的安全性同样是基于有限域的离散对数问题，故而其攻击也都

是尝试去解决离散对数问题，即DLP。常见的攻击算法有Brute Force，Pollard-pho，Pohlig-Hellman，indexcalculus。但是对于DSA，p一般都在512位以上，攻击需要花费大量的时间，而且不一定能求出私钥x。所以，通常也可以采取“Patch p”的方法，即替换p及DSA系统的其他参数，从而达到破解的目的。

6.3.4 椭圆曲线密码编码学（Elliptic Curve Cryptography）

椭圆曲线（Elliptic Curve）作为代数几何学中一个重要问题已经有一百多年的研究历史，但直到1985年N. Kobitz和V. Miller才分别独立地将椭圆曲线引入密码学。由于其相比于RSA等公钥算法，在使用较短的密钥长度而能得到相同程序的安全性，而使得椭圆曲线密钥学的应用越来越广泛，对其的研究也如火如荼，预测未来ECC（Elliptic Curve Cryptography）将取代RSA成为主流的公钥算法。

对于椭圆曲线的完整的数学描述已超出本书的范围。本书只介绍基于GF(p)上的椭圆曲线及ECC在软件保护中的应用所需要的基本知识，$GF(2^m)$上的椭圆曲线本书不作介绍，读者可以参阅其他资料。

1．基本概念

（1）群

阿贝尔（Abelian）群（G,*）由集合G和二进制操作*组成：$G\times G\rightarrow G$满足如下属性：

- （结合律）a*（b*c）=（a*b）*c，a,b,c∈G
- （存在幺元）存在元素e∈G，使得对所有的a∈G都有a*e = e*a = a
- （存在逆元）对每个a∈G，存在b∈G，使得a*b = b*a = e，b叫做a的逆元
- （交换律）a*b = b*a，a,b∈G

群上的操作常称为加法（+）或者乘法（·）。对于前者，相应的群叫做加法群，加法幺元常用0表示，a的加法逆元用-a表示。对于后者，相应的群叫做乘法群，乘法幺元常用1表示，a的乘法逆元用a^{-1}表示。当G为一个有限集时，称群是有限的，即有限群，同时G中的元素个数称为G的阶（order of G）。

例如，令p为一个素数，并且令$F_p = \{0,1,2,\ldots,p-1\}$表示模p的整数集。那么（$F_p$,＋）表示一个阶为p的有限加法群，其加法幺元为0，＋操作定义为模p的整数相加。又如，[image]表示一个阶为p - 1的有限乘法群，其乘法幺元为1，[image]表示F_p中的非零元，·操作定义为整数模p相乘。

若G是一个阶为n的有限乘法群且g∈G，那么使得$g^t = 1$的最小正整数t叫做g的阶。通常t存在且是n的一个因子。集合$\langle g\rangle = \{g^i, 0\le i\le t-1\}$是同G具有相同操作的群，叫做由g生成的G的循环子群（cyclic subgroup of G generated by g）。对于加法群，g的阶是使得tg = 0的最小正整数t，且t是n的一个因子，集合$\langle g\rangle = \{ig, 0\le i\le t-1\}$。这里tg表示加t个g。如果G有一个阶为n的元素g，那么G叫做循环群且g叫做G的生成元（generator of G）。

（2）椭圆曲线群

令p为素数，F_p为模p的整数域。F_p上的椭圆曲线E用如下形式的等式来定义：

[image]

其中$a,b\in F_p$，且满足$4a^3+27b^2\neq 0$（mod p）。一个有序偶（x,y），若x,y满足式（6-10），则称（x,y）为椭圆曲线上的一个点。无穷远点，用∞表示（或者用大写的字母O表示），也在椭圆曲线上。椭圆曲线E上所有点组成的集合记作E（F_p）。

例如E是定义于F_7上的椭圆曲线，其方程为：

[image]

那么E上的点为：

[image]

（3）椭圆曲线密钥的生成

设E是定义于有限域F_p上的椭圆曲线。设P是E（F_p）中的一个点，假设P有一个素数阶n，那么由P生成的E(F_p)的循环子群为：

[image]

素数p、椭圆曲线E、点P及其阶n构成公共参数。私钥是随机从区间[1,n - 1]选取的一个整数d，且其相应的公钥为Q = dP。给出公共参数及公钥Q，求解d的问题叫做椭圆曲线离散对数问题（Elliptic Curve Discrete Logarithm Problem, ECDLP）。

（4）简单的椭圆曲线加密体系

用椭圆曲线上的一个点M表示明文m，Q是接收者的公钥，k为一随机选取的整数，然后通过计算M + kQ来对m进行加密。发送者将$C_1 = kP$和$C_2 = M + kQ$传输给接收者，接收者利用私钥d计算：

[image]

然后就可以恢复M了，$M = C_2 - dC_1$。

（5）椭圆曲线上的运算

令E是定义在域K上的椭圆曲线。E(K)上的两个点相加定义为椭圆曲线上的加法操作。点集E(K)及加法操作构成了一个阿贝尔群，无穷远点∞为加法幺元。这个群用来构造椭圆曲线加密系统。加法操作可以用几何图形来很好地解释。令P =（x_1,y_1），Q =（x_2,y_2）是椭圆曲线E上的两个不同的点。P与Q的和R定义如下：通过P和Q作一条直线，与椭圆曲线交于第三点，那么R与第三点关于x轴对称，如图6.6所示。

[image]

图6.6　R = P + Q

P的两倍R按如下规则定义：过点P作椭圆曲线E的切线，与椭圆曲线E交于第二点，那么R就是第二点关于x轴的对称点，如图6.7所示。

[image]

图6.7　P + P = R

定义于F_p（p为大于3的素数）上的椭圆曲线E的运算规则还包括：

① 幺元：对任意的$P \in E(F_p)$，$P + \infty = \infty + P = P$。

② 逆元：若$P =（x,y）\in E(K)$，那么（x,y）+（x,-y）= ∞。点（x,-y）叫做点P（x,y）的逆，记作-P。实际上-P也是椭圆曲线上的一个点。

③ 点相加。令$P =（x_1,y_1）\in E(F_p)$，$Q =（x_2,y_2）\in E(F_p)$，且$P \neq \pm Q$，那么$P + Q =（x_3,y_3）$。其中：

[image]

④ 点倍乘。令$P =（x_1,y_1）\in E(F_p)$，且$P \neq -p$，那么$2P =（x_3,y_3）$。其中：

[image]

例：（定义于有限域F_{29}上的椭圆曲线）令p = 29，a = 4，b = 20，则椭圆曲线方程为：

[image]

椭圆曲线E上的点为：

[image]

例如：（5,22）+（16,27）=（13,6）,2（5,22）=（14,6）

证明：令P =（5,22），Q =（16,27），则

[image]

上面的证明过程中涉及模29的乘法逆元的计算，如

$(16 - 5)^{-1} \bmod 29 = 11^{-1} \bmod 29 = 8$，即$8 \times 11 \equiv 1 \bmod 29$。

（6）椭圆曲线离散对数问题

离散对数问题的困难性是所有椭圆曲线密码体系安全性的必要保障。

椭圆曲线离散对数问题定义：给定一个定义于有限域F_q上的椭圆曲线E，一个阶为n的点P，且$P \in E(F_q)$，点$Q \in \langle P \rangle$，求出使Q = lP成立的区间[0,n - 1]内的整数l。整数l叫做Q以P为底的离散对数，记作$l = \log_P Q$。

用于密码学的椭圆曲线参数需要仔细地选取以抵抗对ECDLP的所有已知攻击。解决ECDLP的最基本的算法是穷举搜索，即计算点序列P,2P,3P,4P,…直到结果为点Q。但是当n较大时，这种方法是不实际的。其他的攻击方法还有Pohlig-Hellman算法、Pollard's pho算法、index-

calculus算法、Isomorphism算法。最好的已知普通意义上对ECDLP的攻击是Pohlig-Hellman和Pollard's pho算法。本书不是专门讨论椭圆曲线密码的，故不对这些攻击算法的实现进行介绍，感兴趣的读者请进一步参阅相关资料。需要注意的是，没有数学证明ECDLP问题是难于解决的，但是也没有人证明存在有效的算法解决ECDLP。

椭圆曲线在密码协议中可以用于数字签名、公钥加密和密钥交换协议。基于椭圆曲线的数字签名体系如ECDSA、EC-KCDSA，公钥加密体系如ECIES、PSEC，密钥交换协议如STS、ECMQV。

2．椭圆曲线数字签名算法ECDSA

椭圆曲线数字签名算法ECDSA全称为The Elliptic Curve Digital Signature Algorithm，它类似于数字签名算法DSA。它是标准化最为广泛的基于椭圆曲线的签名体系，在ANSI X9.62、FIPS 186-2、IEEE 1363-2000及ISO/IEC 15946-2等标准中均有说明。

（1）算法描述

域参数D＝（q,FR,S,a,b,P,n,h）由以下几部分组成：

① 域的阶q，即椭圆曲线上点的个数$\#E(F_q)$；

② 有限域F_q上的元素的域表示FR（field representation）；

③ 如果椭圆曲线是随机生成的，那么S代表种子；

④ 系数$a,b \in F_q$，其定义了F_q上的椭圆曲线E：$y^2 = x^3 + ax + b$；

⑤ 定义两个域元素x_P和y_P，其定义了点P＝（x_P,y_P）$\in E(F_q)$，$P \neq \infty$，且P有一素数阶，P叫做基点（Base Point）；

⑥ P的阶n；

⑦ 余因子$h = \#E(F_q)/n$。

（2）ECDSA签名生成算法

在如下的表述中，H代表散列函数，通常为SHA-1。

输入：域参数D＝（q,FR,S,a,b,P,n,h），私钥d，待签名的消息m

输出：签名（r,s）

① 随机选择$k \in [1,n-1]$；

② 计算kP＝（x_1,y_1），并且将x_1转化为整数[image]

③ 计算[image]如果r＝0，则转到步骤①重新生成签名；

④ 计算e＝H(m)。

⑤ 计算$s = k^{-1}$（e＋dr） mod n。如果s＝0，则转到步骤①重新生成签名；

⑥ 返回（r,s）。

（3）ECDSA签名验证算法

输入：域参数D＝（q,FR,S,a,b,P,n,h），公钥Q，待验证签名的消息m，签名（r,s）

输出：签名有效或者无效

① 验证签名r和s是否为区间[1,n－1]内的整数。如若不是，则返回“签名无效”；

② 计算e＝H(m)；

③ 计算$w = s^{-1} \bmod n$；

④ 计算$u_1 = ew \bmod n$及$u_2 = rw \bmod n$；

⑤ 计算$X = u_1P + u_2Q$；

⑥ 如果$X = \infty$，则返回“签名无效”；

⑦ 将X的x轴坐标x_1转化成整数[image]并计算[image]

⑧ 如果$v = r$，则返回“签名有效”，否则返回“签名无效”。

如果消息m的签名（r,s）是由合法的签名者生成的，那么$s \equiv k^{-1}(e + dr)(\bmod n)$，即：

[image]

所以$X = u_1P + u_2Q = (u_1 + u_2d)P = kP$，因此只要$v = r$即可验证签名是有效的。

3．ECDSA在软件保护中的应用

目前在一些商业保护中使用了ECDSA算法来作为注册验证的主要部分，如Safecast、Flexlm。ECDSA的实现还涉及许多关于如何选取合适的椭圆曲线，选取安全的基点（Base Point）等一些知识，这些知识需要相关的数学知识，本书不做介绍，感兴趣的读者可以参考相关资料。

下面使用miracl给出一个实现如何使用ECDSA来作为软件的序列号生成及验证机制的例子。本例中的椭圆曲线参数采用NIST在“Recommended Elliptic Curves for Federal Government Use”中推荐的GF(p)上椭圆曲线之一Curve P-192。

对应于域参数D =（q,FR,S,a,b,P,n,h），本例中的参数如下：

[image]

随机生成的160位的私钥d为：

[image]

公钥Q为$Q = (Q_x,Q_y)$，其中：

[image]

（1）签名的生成

首先计算消息（用户名）的160位SHA-1散列值。

[image]

紧接着初始化大数，此时可以直接将椭圆曲线的参数之一a初始化。

[image]

然后先初始化椭圆曲线。

[image]

ecurve init的第4个参数有两个取值，定义如下：

[image]

这两个参数主要用来指定系统内部椭圆曲线上点的坐标的表示方法，通常用MR_PROJECTIVE，因为使用这种坐标表示方法时，速度会更快。

在生成签名的过程中，要保证随机数k、签名r、签名s都不能为0，如果为0，则需要重新生成签名，所以使用while循环。代码如下：

[image]

[image]

上面的代码段中使用了许多的miracl中的函数，读者可以参考miracl的手册查看其参数的详细说明。这里只强调函数epoint_set，其函数原型定义如下：

[image]

参数x，y分别为点P的x坐标和y坐标。第三个参数lsb涉及椭圆曲线上点压缩的概念。点压缩是实现椭圆曲线的一种技术，它可以降低椭圆曲线上点的存储空间。对于椭圆曲线$y^2 = x^3 + ax + b$上的点（x,y），给定一个x，y可能有两个值与之相对应，这可以从前面的例子中看出来。这是因为椭圆曲线上点的运算是模点的阶n的，当计算出来的y为负值时，需要重复加上n，直到y为正，这样椭圆曲线上就存在两个点，具有相同的x坐标，其中一个是奇数，即其最低位（least significant bit）为1，另一个是偶数，其最低位为0。因此只要给出x坐标的值和y坐标的最低位的值0或1就可以确定一个点。

（2）签名的验证

验证的第一步是确保签名r，s必须位于区间[1,n - 1]内，否则签名无效。

[image]

紧接着要验证点X是否为无穷远点，如果是无穷远点，则签名无效。

[image]

更为详细的代码请参考光盘中的ECDSASample。

6.4　其他算法

除了以上算法外，平时经常接触的还有CRC32算法、Base64编码等算法。

6.4.1　CRC32算法

CRC全称为“Cyclic Redundancy Checksum”或者“Cyclic Redundancy Check”，是对数据的校验值，中文名是“循环冗余校验码”，常用于校验数据的完整性。最常见的CRC是CRC32，即数据校验值

为32位。

首先利用CRC32多项式的值04C11DB7h或者EDB88320h（将CRC32多项式的二进制表示的字符串逆向计算即可得到此值）生成一张CRC32表，其算法用C代码表示如下：

[image]

生成具有256个元素的CRC32表，如表6-4所示，完整的见光盘。

表6-4　CRC32数据表

[image]

然后就可以根据CRC32数据表来计算字符串或者文件的CRC32值了。算法如下所述：

[image]

有关CRC32在文件完整性校验的应用请参考第14章。由于CRC32代码量小，容易理解，所以目前应用十分广泛。但同时，CRC32并不是一个安全的加密算法。如果需要更安全的完整性校验算法，建议使用数字签名技术。

6.4.2　Base64编码

Base64编码是将二进制数据编码为可显示的字母和数字，用于传送图形、声音和传真等非文本数据。常用于MIME电子邮件格式中。其使用含有65个字符的ASCII字符集（第65个字符为“=”，用于对字符串的特殊处理过程），并用6个进制位表示一个可显示字符。表6-5列出的就是Base64编码表。

表6-5　Base64编码表

[image]

把数据编码为Base64，将第一个字节放置于24位缓冲区的高8位，第二个字节放置于中间的8位，第三个字节放置于低8位。如果对少于3个字节的数据进行编码，相应的缓冲区位将被置0。然后对24位缓冲区以6位为一组作为索引，高位优先，从字符串“ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz01234
取出相应的元素作为输出。如果仅有一个或两个字节输入，那么只使用输出的两个或三个字符，其余的用“=”填充。

解码是编码的逆过程。首先得到Base64字符串的每个字符在Base64码表中的索引，然后将这些索引的二进制连接起来，重新以8位为一组进行分组，即可得到源码。

例如，表6-6里“转换前”栏的三个字节是原文，“转换后”栏的四个字节是Base64编码，其前两位均为0。

表6-6　Base64值编码

[image]

除了Base64以外，还有Base24、Base32和Base60。

Base24码表：

BCDFGHJKMPQRTVWXY2346789（Windows产品序列号就是使用这种编码）

Base32码表：

ABCDEFGHIJKLMNOPQRSTUVWXYZ234567

Base60码表：

0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrst

需要注意的是，在实际使用中，码表会和这些标准的码表不一样，但其编码原理是一样的。

6.5　常见的加密库接口及其识别

程序员在自己的程序中实现加密算法时，往往需要借助于一些加密算法库来实现。逆向分析时必须能识别出常见的加密算法库，识别加密算法库，最直接也是最精确的方法，便是掌握该算法库的使用方法。另外，也可使用IDA的Flair工具制作出算法库的signature。

6.5.1　Miracl大数运算库

Miracl全称为Multiprecision Integer and Rational Arithmetic C/C＋＋ Library，即多精度整数和有理数算术运算C/C＋＋库。它是一个大数库，实现了设计使用大数的加密技术的最基本的函数。支持RSA公钥系统、Diffie-Hellman密钥交换、DSA数字签名系统及基于GF(p)和$GF(2^m)$的椭圆曲线加密系统。Miracl提供了C和C＋＋两种接口，而且使用起来非常方便，速度相当令人满意，并且是开源的，所以用户可以根据需要扩充这一优秀的加密算法库。

官方网站为www.shamus.ie。下载完最新版本后，参考readme.txt将其安装好。本节开发环境为Visual C＋＋6.0，使用C接口。为了方便，可以先将miracl目录下的include文件夹中的*.h头文件统一复制到VC6（VC98）的include目录中，并且将miracl编译好的静态链接库ms32.lib放到VC6（VC98）的lib目录中，这样，在以后使用miracl的工程中，只需要添加下面两个语句即可调用miracl中的函数了。

[image]

以光盘中的MiraclStudy.exe为例，介绍miracl中的大数格式及其函数的识别。

1．miracl中的大数格式

miracl中的大数是以2^{32}进制来表示大数的，即当数字大小超过FFFFFFFFh时，向高位进1。在miracl.h中，miracl是如下定义大数的：

[image]

可见在程序中定义一个大数变量时，如big bigN，那么实际上是定义了一个结构指针，结构的第一个元素是大数的长度，第二个元素是整型指针，指向存储大数数值的内存地址。

用OllyDbg打开光盘中的MiraclStudy.exe，通过跟踪调试可以来到如下代码处：

[image]

40D03C处指向的是一个字符串：

[image]

EDI是一地址，指向一大数变量，在内存中的格式如下：

[image]

第一个双字为00000000h，表示大数的长度为0，第二个双字为003C6C04h，它指向大数的实际数值。在经过402BC0这个函数之后，3C6BF8内存地址处的数据就变成了如下：

[image]

第一个双字为00000008h，表示大数长度为8个双字，即占用了32个字节。从003C6C04地址处开始才是大数的真正数值，可以看出miracl将字符串前面的字符作为低位，采用低位在前，高位在后的形式来存储大数。

2．识别**miracl**中的函数

通过前面介绍的方法，可以通过做出miracl的IDA sig来识别函数。这里介绍另外一种方法：magic number。

在miracl的几乎每个函数的实现中，都有这样一条语句：

[image]

它是miracl的错误处理机制，用来表示miracl中的函数及退出代码。其定义在miracl.h中，通过它miracl可以知道是哪个函数出了错误。每个函数的“xx”都不同，如函数mirvar为MR_IN(23)，函数set_user_function为MR_IN(111)。通过反汇编，可以发现，这句代码通常都是如下形式：

[image]

这里的yy是上面的xx的十六进制形式。根据yy就可以判断此函数究竟是miracl中的哪个函数了。笔者整理出了miracl 5.01版本中的这些函数的“magic number”，见光盘。

用OllyDbg打开光盘中的MiraclStudy.exe，通过调试跟踪可以来到

如下代码处：

[image]

通过上面的代码，17h即十进制23，可以判断出程序调用的是miracl中的mirvar函数。通过类似的方法，继续向下跟踪，可以分别找到如下的代码：

[image]

可以判断出程序接着分别调用了bytes_to_big，cinstr，powmod，cotstr四个函数。

6.5.2 FGInt

FGInt全称为Fast Gigantic Integers，是用于Delphi的一种可以实现常见的公钥加密系统的库。官方网站是：http://www.submanifold.be，读者可以从其下载最新版本的FGInt。

1．FGInt中的大数格式

FGInt是以2^{31}进制来表示大数的，即当数值超过7FFFFFFFh时，便向高位进1。FGInt大数库在FGInt.pas中定义了大数格式，如下：

[image]

TFGInt为一Record类型，第一个双字是符号位，表示大数的符号；第二个双字是一个LongWord数组，存储了大数的数值。

光盘中的FGIntStudy/Project1.exe是一个使用FGInt实现ElGamal签名的程序。

用OllyDbg打开FGIntStudy.exe，通过跟踪调试可以来到如下代码处：

[image]

004565EC处是一个十进制整数字符串：

[image]

[ebp-30]内存地址处在调用00454290这个函数之前的数据为：

[image]

在经过454290这个函数调用之后，内存数据变为：

[image]

第一个字节01表示的是大整数的符号为正，第二个双字，即00D65378（注，以读者实际数据为准），指向的是存储大数内存地址。查看D65378h地址处的数据为：

[image]

其第一个双字表示大数的长度，占用了5个双字空间，从第二个双字开始才是大数的数值。

2．识别FGInt的函数

第一种方法是先制作出FGInt的IDA signature，然后应用signature即可识别FGInt中的函数。第二种方法是观察函数的参数个数，以及函数调用前后数据的变化来判断此函数的功能，然后对照FGInt的源码，对反汇编代码进行对比，来确定是何函数。第三种方法，对于FGInt，Kanal（PEiD的Kcrypto ANALyzer插件）可以识别出其函数，如本例，用PEiD 0.94中的Kcrypto ANALyzer插件可以识别出如下函数，如图6.8所示。

[image]

图6.8　PEiD插件识别出FGInt

根据其地址即可找到该函数，这样对分析程序具有一定的帮助。

6.5.3　其他加密算法库介绍

1．freeLIP

freeLIP最初设计是用于进行RSA-129挑战大数计算的大数库。其版本为1.1。其接口的详细说明请参考readme及lip.h。

freeLIP采用2^{30}进制来表示大数。其速度不及miracl。

2．Crypto＋＋

Crypto＋＋是一个实现了相当数量的加密算法的加密库，由华人戴伟开发。其官方网址为：http://www.cryptopp.com。Crypto＋＋使用了C＋＋的高级语法，再加上文档比较少，所以不容易上手。

Crypto＋＋的应用十分广泛。对于其函数的识别目前还没有很好的办法，常用的是做出其IDA sig，然后应用sig去识别其函数。但是由于该库实现了相当数量的函数，所以在做sig时会有大量的冲突，需要花不少时间，但是可以通过编写脚本自动处理。而且由于不同的用户在编译该库时，采用的编译器设置不同，会造成sig不能完全识别其函数的情况。根据笔者的经验，识别其函数依靠于对该库的熟悉程度，依赖于对加密算法的掌握程度，如果对加密算法的加密原理及其实现过程相当熟悉，那么不论程序采用了何种加密算法库，都可以轻而易举地识别出。

3．LibTomCrypt

LibTomCrypt是另一款相当不错的加密算法库，其包括了常见的散列算法、对称算法及公钥加密系统。官方主页：http://libtomcrypt.org。其接口相当友好，非常适合C程序员使用。而且其代码书写相当规范，使用很方便。

其函数的识别，最有效的办法是，做出其IDA sig，应用sig便可识别

出其大部分的函数。

4．GMP

GMP全称为GNU Multiple Precision Arithmetic Library，其核心采用汇编实现，速度非常快，超过miracl。常用其来实现大整数因子的分解，以提高速度。少见使用此库做软件保护。其官方主页为：http://www.swox.com/gmp。

5．OpenSSL

OpenSSL主要用于网络安全，其中也包含了一些加密算法的实现。如对称算法中的BlowFish、IDEA、DES、CAST，公钥中的RSA、DSA，散列中的MD5、RIPEMD、SHA等。官方主页：http://www.openssl.org。

其函数的识别相对来说比较简单，一是通过反汇编，可以在使用了OpenSSL的程序中找到标记OpenSSL版本号的字符串，进而下载相应的版本进行编译，做出其IDA sig，应用sig来识别。二是由于其实现的算法有限，故可以根据反汇编观察函数的参数的个数、参数的类型，然后根据这些信息到OpennSSL的crypto目录下的加密算法源代码中去找符合条件的函数，然后一一进行排除，甚至通过写几个测试程序，来检测函数的功能。

6．DCP和DEC

DCP全称为Delphi Cryptography Package，是用于Delphi的一个加密算法库。官方主页：http://www.cityinthesky.co.uk/cryptography.html。

DEC全称为Delphi Encryption Compendium，同样是用于Delphi的一种加密算法库。光盘中提供了其软件包。

由于这两个加密算法库实现了大部分常见的散列算法及对称算法，使用也十分方便，故而经常被Delphi程序员用于其软件的保护中。具体的使用请参考其文档。

对于DCP和DEC中函数的识别，最有效的方法是使用其IDA sig。其次，取决于读者对加密算法的熟悉程序，也可以很容易地识别其加密函数。

7．Microsoft Crypto API

Crypto API是微软为了方便程序员在软件中进行数字签名、数据加密的开发而提供的一套加密系统。接口也相当友好方便。其详细的说明请参考msdn。

由于是微软提供的API，故IDA及Ollydbg、Softice等调试软件都可以识别其函数。

8．NTL

NTL是一个可以用于数论相关计算的库。官方主页：http://shoup.net/ntl/。它提供了非常友好的C＋＋接口，用于实现有符号的、算术整数的运算，以及向量、矩阵、基于有限域和整数的多项式运算。

在密码学中，有限域的应用相当广泛，如AES、twofish、ECC等都涉及有限域。感兴趣的读者可以参考数论相关资料。

同样，只要做出该库的IDA sig，即可达到识别其函数的目的。

注释

① 本章由沈晓斌编写。

第4篇　语言和平台篇

- 第7章　**Delphi程序**
- 第8章　**Visual Basic程序**
- 第9章　**.Net平台加解密**

必须根据编译语言的特点进行加密与解密，才能达到比较理想的效果。现在所使用的语言无非是两种：一种是解释执行的语言，另一种就是编译后才能够执行的语言。解释语言的最大弱点之一就是能被反编译，因此其保护的重点应放在如何防止反编译上。

第7章　Delphi程序

Delphi与C＋＋ Builder同为Inprise公司产品，共享集成开发界面（IDE），而且使用同一套VCL（Visual Component Library）框架。事实上，Delphi和C＋＋ Builder除了使用的语言不同，其余几乎都相同，因此本节只以Delphi为例来讲述。

Delphi和C＋＋ Builder采用的是VCL（可视化组件库）技术，这些环境使用特有的资源格式RCDATA。RCDATA资源中含有Delphi的窗体（Forms），所有对窗口设计的信息都包含在内。当一个典型的Delphi程序开始运行时，其初始化代码建立这种窗体，并从资源中读取所需要的信息。DFM文件是Delphi编码的二进制文件。在编译时，这个文件就以源程序的形式存储于RCDATA资源中。这样，利用eXeScope等资源编辑工具，可以对它查看、修改，并且可改变程序的某些运行方式。

7.1　DeDe反编译器

Delphi/C＋＋ Builder采用控件拖放的方式来完成界面的设计，并和事件联系起来。而这些信息以资源（RCDATA）的方式存放于可执行文件中。DeDe便利用这个原理进行反编译，获取相关信息，将界面与事件联系关系还原，但事件的汇编代码不能还原。DeDe公开了源代码，感兴趣的读者可以研究一下。

1．主要功能

用DeDe可以查看Delphi程序窗体的属性，可以查看按钮对应的事件，并将事件代码反汇编出来，其能识别出Delphi库函数，具有良好的可读性。另外，还可以把事件输出到map文件中供其他工具使用。

2．配置

（1）DSF文件

① DSF文件的含义

DSF文件内容来自不同版本BPL库文件的输出符号表。DeDe反汇编引擎使用这些符号表对生成的ASM代码文件添加类成员方法调用的注释，这非常类似于IDA Pro的FLIRT技术。如果没有加载任何BPL符号表文件，对BPL类的调用就无法以注释的格式说明。

② 加载DSF文件

经由“File/Load Symbol File”菜单，就可加载所需要的DSF文件。程序若能正确识别出相应版本的Delphi程序，会自动加载DSF文件。若希望每次启动DeDe的同时自动加载若干DSF文件，选中“Options/Configuration”菜单，在Symbols选项卡中就可完成本项工作。如果想查看包含在某个特定DSF文件中的输出符号表，选择“Options/Symbols”。

③ 为何需要创建DSF文件

处理使用自定义构件（即不是Delphi安装的构件）的程序时，如果有这些自定义构件的BPL，并且为它们创建了DSF文件，那么DeDe将会注释所有对这些自定义构件的调用。创建DSF的速度也是很快的。

（2）DOI文件

DOI意思是Delphi Class Offset Information（Delphi类偏移信息），该技术使用偏移信息识别类成员：方法和域（实例变量，属性）。DOI文件包含用于识别的必要数据。DeDe仿真指令的执行来查找使用这些偏移的引用所在。例如，在继承自TForm类的任何子类的偏移0xCC处，代表指向ShowModal方法的指针。当遇到类似call[reg + $00CC]的调用时，仿真器就知道寄存器包含的对象是引用TForm.ShowModal方法的TForm子类。DOI文件应该存放在DSF文件夹内。

下面是一个生成的带有DOI帮助信息的简单示例。

[image]

使用DOI文件，只需复制*.DOI文件到DSF文件夹即可。DOI数据会自动插入到生成的代码文件中。

（3）字符串参考的含义

在DeDe中，如果处理含有非英文字符串的程序，则选择“Option/Configuration”菜单，在References选项卡中可以设置DeDe反编译引擎查找字符串参考时使用的字符集。

[image]注意：如果使用全部的字符集合#32～#255，则可能得到残缺的字符串参考。Delphi程序一般不用Unicode字符串，这也是该选项没有包括在字符串参考配置中的原因。

3．基本操作

DeDe安装很简单。安装好后直接执行主程序，出现如图7.1所示的主界面。单击[image]按钮打开光盘映像文件中的DE_Delphi文件，然后单击“Process”按钮进行反编译。DeDe先将被分析的文件装载到内存中，再进行反编译，因此对一些压缩加壳的程序也能反编译。

[image]

图7.1　DeDe界面

• CLasses Info：显示程序中使用的类信息；

• Units Info：显示程序中使用的单元信息；

• Forms：显示程序中的窗体信息，这部分可以用资源编辑工具修改；

• Procedures：显示程序的过程信息；

• Project：可将当前项目保存；

• Exports：导出符号文件。

在此显示了 Events（事件）和Controls（控件）两方面的内容（见图7.2）。Button1Click事件对应的是“确定”按钮，双击可打开代码窗口。该窗口显示当前事件对应的汇编代码，右边控制条显示全局变量和局变量。双击某个表达式，可以加注释。在跳转指令、CALL指令上双击可跳到相应代码上。

[image]

图7.2　查看事件按钮

设目标实例DE_Delphi的用户名为Name[]，具体代码如下：

[image]

[image]

由于DeDe的符号识别技术，使得上述代码可读性非常容易理解。

在图7.2中有两个按钮值得注意：

• DPR按钮：是反汇编项目文件.dpr，该文件控制或记录程序中的所有文件。

• OFFS按钮：反汇编指定地址的代码。

7.2　按钮事件代码

动态调试Delphi程序时，会发现用一般的API函数不能拦截文本框（Edit）的数据。因为Delphi通过向Edit发送WM_GETTEXT消息来获得Text的内容（直接调用WndProc，而没有使用消息函数），整个过程没有调用过任何Win32 API函数，所以常用的Hmemcpy，GetDlgItemTextA，GetWindowTextA等断点失效是当然的。那么，如何才能将用户输入的字符串拷贝到软件的缓冲区中时使调试器中断呢？可以利用DeDe来辅助，Delphi的按钮和事件对应，利用DeDe反编译得到按钮的事件地址，对此地址设断拦截。本节讨论一下如何手工定位按钮的事件代码。用eXeScope打开DE_Delphi文件，查看其RCDATA资源，找到“确定”按钮的资源。数据如下：

[image]

例程（用来响应用户界面之窗体的元素）的地址是按名字绑定的。只要知道这些名字，就可知道所需要的地址，因此搜索该例程名称就能找到调用的地址。用十六进制工具打开DE_Delphi，搜索“Button1Click”，找到两处。第二处是资源本身，第一处是在地址表（Address Table）中，具体见图7.3。

[image]

图7.3　查看OnClick事件地址

现在，观察名字前那些神奇的数字，有一个字节“0C”指出了“Button1Click”的长度（12个字符）。在此字节前面的就是其事件调用地址4502A4h，然后就可用此地址设置断点了。

该方法不仅适合寻找按钮的调用地址，也适合寻找FormCreate等过程事件的地址。找到后，可以用图7.2中的“OFFS”按钮对这些指定地址进行反汇编。

7.3 模块初始化与结束化

Delphi采用的是VCL体系，它的类是由编译期间决定的，编译完成后即固定在程序中。当程序运行时，首先会将这些类完成初始化，然后开始执行代码。

用OllyDbg打开实例DE_Delphi_6.exe，以分析程序是如何初始化的。下面这段代码是Delphi 6编译器为可执行程序EXE生成的入口代码：

[image]

注意这句“00450662 mov eax, 4504EC”，4504EC负责传入单元的初始化表地址，其指向的是初始化表的个数与地址，如图7.4所示。初始化表的个数是2Dh，地址是4504F4h。

[image]

图7.4 初始化表大小与地址

InitExe()处理初始化表的相关代码位于Delphi安装目录下的SysInit.pas，System.pas里，代码如下：

[image]

相应的汇编代码如下：

[image]

跟进StartExe()函数，代码如下：

[image]

相应的汇编代码如下：

[image]

InitUnits就是调用初始化表的代码，其主要的汇编代码如下：

[image]

从上面代码分析可以得知，Delphi程序加载时，程序从前往后间隔调用初始化表，退出是从后往前间隔调用的。如图7.5所示，程序运行时，会间隔地调用4065EC，406440，406664…。退出时，从后往前间隔调用，即从地址00450658开始向前，如4504C4，45048C，450000…。所以图7.4中的初始化表个数2Dh是整个初始化表的一半，整个初始化表个数是2Dh×2。

[image]

图7.5　Delphi的初始化表

程序的入口点也就在初始化表后，在后面章节脱壳技术中，可以用这种方法确定入口点。同时一些壳（如ASProtect）加密了初始化表，需要还原。

第8章　Visual Basic程序

VB3和VB4是典型的解释语言，它们都有相应的反编译器存在。VB5和VB6不再是单纯的解释程序，在继续保留P-code编译的同时也引入了Native编译方式，使得生成本地二进制代码成为可能。关于VB.net，其生成的EXE程序完全可用.Net反编译工具完成，此处不再介绍。

本章仅简单地介绍Visual Basic程序一般的调试方法，有关Visual Basic 6逆向工程更多的知识，请参考《软件加密技术内幕》此书周文雄撰写的“Visual Basic 6逆向工程”。

8.1　基础知识

8.1.1　字符编码方式

Visual Basic 32位版本的字符串处理采用Unicode编码或称宽字符（Widechars）。也就是说，字符串在Visual Basic内部是以Unicode格式存放的。在Unicode中，所有的字符都是16位的，比如7位ASCII码都被扩充为16位（注意：高位扩充的是零，即0）。

Visual Basic中的字符串因为版本不同，字符串的格式也不同。16位VB 4.0和以前版本的VB字符串格式很复杂，使用连续指针的方式，源指针指向一个由字符串长度和字符串首地址指针组合而成的结构，其中的字符串首地址指针再指向字符串。VB 5.0及以后版本则是单指针方式，源指针指向一个复合字符串，此字符串开始的四个字节组成一个长整数，以指示此字符串的长度，其后是字符串，而且这个字符串除了应有的字符外，结尾还有两个“00”，而最后的这两个“00”不计算在字符串长度之内。一个英文的VB字符串“pediy”在内存中（编译后）应该如图8.1所示。

[image]

图8.1　VB字符串在内存中的形式

8.1.2　编译模式

编译器的编译技术可以分为Native-Compile（自然编译）与Pcode-Compile（伪编译）两种。自然编译是编译器将高级语言转换为汇编代码，并经链接生成EXE程序的过程。伪编译是编译器将高级语言转换为某种编码后，将能解释、执行此编码的一段程序一同链接，生成EXE程序。

所谓伪代码，其基本工作原理是编译器先把执行程序编译成比80x86机器码紧凑得多的中间代码形式，运行时，其基于堆栈的引擎将特殊操作码翻译成CPU上的操作指令。依靠P-code编译技术，使得编程语言不依赖于机器或系统平台成为可能。

从Visual Basic版本1.0到4.0都只生成P-code。P-code的最大问题就

是执行速度比本地编译的程序慢，优点是使得程序不依赖于硬件或操作系统成为可能。从1997年Visual Basic 5问世起，微软公司在编译过程添加了允许生成本地机器码的选项。虽然目前P-code编译形式大多见于Visual Basic，但Java、PowerBuilder等的编译实质也是一样。

因为VB3和VB4都是使用伪代码编译形式，也因为伪代码编译的代码实际只是“变形的源代码”，所以只要能理解其对应机制，就能做出反编译器来。例如Dodi's VB 3/4 Disassembler。同样，Java、PowerBuilder也有反编译器。

8.2 自然编译（Native）

自然编译是VB源代码生成汇编代码并链接的过程。因为是生成汇编代码，所以对其解读完全遵照一般的反汇编常规。

8.2.1 相关VB函数

掌握一些VB常用内部函数，将有助于VB程序的分析。以VB 6.0为例，实际上的数据计算和比较是在msvbvm60.dll及oleaut32.dll中完成的。如果要深入研究VB，不妨用IDA、W32Dasm反汇编msvbvm60.dll及oleaut32.dll，理解其函数的意义。msvbvm60.dll中包含的函数名总是用_vba或rtc开头，而oleaut32.dll函数名以var开头。对于函数的作用，一般可以按照函数名从右往左理解。例如：_vbaI2Str，表示字符串转换为整数。

通过研究OLEAUT.H、OAIDL.h文件可以了解VB函数中各类变量，并进而根据函数名称猜出大部分函数的用途，见表8-1。

表**8-1** 相关函数

[image]

[image]注意：这些函数前的下划线“_”是由两根短线“_”组成的。

8.2.2 VB程序比较方式

本节从汇编角度来分析一下VB字符串的比较模式，以熟悉VB的一些处理方式。

1．字符串（**String**）比较

在这种方法里，正确密码串（如“Correct Password”）和输入的密码串（如“Entered Password”）比较。字符串是由相邻的字符按顺序排列组成的。一个字符串包括字母、数字、空格和标点符号。

下面是一个范例代码：

[image]

这种方式是明码比较。如果程序用这种函数比较，很容易被破解。可用到的断点：_vbaStrComp或_vbaStrCmp。

运行光盘映像文件中用VB 6编译好的实例string.exe，可以用

_vbaStrCmp或rtcMsgBox设置断点。汇编形式如下：

[image]

OllyDbg断下后，在数据窗口查看401BD8处的数据，如图8.2所示。其以宽字符显示，转换过来就是“pediy”，这就是真的序列号。

[image]

图8.2　在数据窗口查看字符

2．变量（Variant）比较

在这种方法中，两个变量（变量数据类型）互相比较。变量数据类型是一种特殊数据类型，包括数字、字符串、日期数据及一些用户定义的类型。这种类型储存的数字长度是16字节，而字符长度是22字节（加上字符串长度）。

下面是一个范例代码：

[image]

在此方法中，字符串比较中的两个API断点将不起作用，因为程序不再用_vbaStrCmp等函数比较字符串。其比较的方式是依赖于变量是否相等，即用_vbaVarTstEq函数比较变量。

用VB 6编译好的程序为Variant.exe。比较代码汇编形式如下：

[image]

在OllyDbg的命令行里，执行D[edx + 8]，将显示出正确的序列号，其以宽字符显示，如图8.3所示。

[image]

图8.3　在数据窗口查看参数

[image]注意：此时直接查看ecx和edx什么都看不到，这是因为VB用了一些特殊的寻址方式。牢记此时用ecx + 8和edx + 8查看内存，将显示真假序列号的偏移地址。

3．字节（Byte）比较

这种方法是两个字节数据进行比较。Byte变量存储为单精度型、无符号整型、8位（1个字节）的数值形式，范围在0至255之间。

下面是一个范例代码：

[image]

对这种类型没有专门的断点函数，因为其数据比较是在主程序里，而不是在Visual Basic运行库中。

用VB 6编译好的程序为Byte.exe。代码用十六进制数据比较，汇编

形式如下：

[image]

4．整型（Integer）比较

这种方法是用两个整型数据进行彼此比较。Integer变量存储为16位（2个字节）的数值形式，其范围为-32768到32767之间。

下面是一个范例代码：

[image]

对这种类型没有专门的断点函数，因为其数据比较是在主程序里，而不是在Visual Basic运行库中。

用VB 6编译好的程序为Integer.exe。代码用十六进制数据比较，汇编形式如下：

[image]

5．长整型（Long）比较

这也是一种常见的方法，两个变量（长整型）互相比较。Long（长整型）变量存储为32位（4个字节）有符号的数值形式，其范围从-2147483648到2147483647。因此该方法只能用来比较数字。

下面是一个范例代码：

[image]

用VB 6编译好的程序为Long.exe。代码用十六进制形式比较，汇编形式如下：

[image]

6．单精度实数（Single）比较

这种方法用两个单精度实数数据比较。Single（单精度浮点型）变量存储为32位IEEE（4个字节）浮点数值的形式，它的范围在负数的时候是从-3.402823E38到-1.401298E-45，而在正数的时候是从1.401298E-45到3.402823E38。因此这种方法仅仅能比较数字，但也可将姓名和序列号转换成实数进行比较。查看相关数据时用DL命令显示浮点型（Long/Real）。

下面是一个范例代码：

[image]

用VB 6编译好的程序为Single.exe。代码用单精度实数数据格式比较，汇编形式如下：

[image]

7．双精度（**Double**）比较

这种方法用两个双精度数据比较。Double（双精度浮点型）变量存储为64位IEEE（8个字节）浮点数值的形式，它的范围在负数的时候是从-1.79769313486231E308到-4.94065645841247E-324，而正数的时候是从4.94065645841247E-324到1.79769313486232E308。与单精度类似，只能比较数字。

下面是一个范例代码：

[image]

用VB 6编译好的程序为Double.exe。由于比较的数是64位，因此分两次比较。第一次是比较双精度的前32位“9C5FE400”，第二次是比较双精度的后32位“42A674E7”。此例中的汇编形式如下：

[image]

8．货币（**Currency**）比较

这种方法是用两个Currency数据类型进行比较。是的，它听起来是怪，但确实存在！Currency变量存储为64位（8个字节）整型数值的形式，然后除以10000给出一个定点数，其小数点左边有15位数字，右边有4位数字。这种表示法的范围可以从-922337203685477.5808到922337203685477.5807。Currency数据类型在货币计算与定点计算中很有用，在这些场合精度特别重要。

下面是一个范例代码：

[image]

用VB 6编译好的程序为Currency.exe。代码将Currency数据分成两段来比较，先比较前32位，再比较后32位。汇编形式如下：

[image]

9．小结

两个表达式都是数值数据类型（Byte，Integer，Long，Single，Double或Currency），进行数值比较。当一个Single与一个Double做比较时，Double会进行舍入处理而与此Single有相同的精确度。如果一个Currency与一个Single或Double进行比较，则Single或Double转换成一个Currency。在实际中，程序一般同时采用两种以上的方法来比较数据，如Currency和String、Variant和Long等。

分析Visual Basic程序时，对Vbrun*.dll（VB 3 和VB 4版本）和Msvbvm*.dll（VB 5和VB 6）强调得比较多。实际上，Visual Basic程序的很多运算是在Oleaut32.dll中完成的，这个DLL提供了对Visual Basic中Variant类型的变量进行操作的许多函数，主要是一系列VarXXX()。

除了用OllyDbg等工具分析VB程序，也可以用SmartCheck辅助分析。SmartCheck是NuMega公司推出的一款针对Visual Basic的错误检测

和调试工具，有关使用参考光盘映像文件中提供的文档。它能够自动检测和诊断VB运行时的错误，并将一些表达不清楚的错误信息转换为确切的错误描述。

8.3 伪编译[1]

伪编译是生成伪代码并链接的过程。运行时依赖解释引擎将伪代码翻译为汇编代码再执行。伪代码的运行完全依赖于堆栈。

如果用IDA或W32Dasm反汇编一下光盘映像文件中提供的vbpcode.exe程序，将会看到莫名其妙的代码，因为它不再是传统意义上的汇编代码了，只有用P-code反编译器才能得到正确的代码。VB 5/VB 6的P-code反编译器现在有Exdec、WKTVBDebugger、VBDE等。

8.3.1 虚拟机与伪代码

相对VB Native Code，P-code编译模式要出现得更早一些。事实上，在5.0以前的版本中，所有的VB程序代码都不加选择地被编译为P-code形式。VB P-code的运行速度较慢，这是由它本身的运行机制所决定的。

1．虚拟执行的原理

P-code，即Pseudo Code（伪代码），这一概念最早出现在Pascal编译器中，它是为了提供跨平台可移植性而产生的，实现这一编译机制的Pascal编译器被称为“Pascal P Compiler”。Sun公司在其推出的Java语言上也成功地实现了这种机制。Java程序的伪编译代码由一系列代表一定意义的字节码（byte code）组成，它们同属于一套特定的指令集，这种字节码不能由不同的CPU直接执行，而是要通过特殊的解释引擎翻译为CPU可以识别的指令才能执行，这种解释引擎就是我们常说的“虚拟机”。只要在不同的平台上提供虚拟机，把字节码翻译为对应的CPU指令集，也就实现了所谓的跨平台特性。

Microsoft推出的VB P-code，实际上也是一组自定义的指令集，必须通过基于堆栈的虚拟机翻译为80x86上的指令集才能执行，担任虚拟机任务的就是系统目录下的msvbvm50.dll和msvbvm60.dll这两个动态链接库文件。由于在文件执行过程中多出了这一个解释的步骤，自然要影响到其执行的速度。至少到目前为止，VB P-code并没有实现所谓的跨平台运行特性，这对于Pseudo这个词的起源是不恰当的：另外，采用P-code形式编译生成的VB应用程序体积一般要小于采用Native Code形式编译的同样程序（这是由于P-code指令集的每一条指令往往对应于一组80×86指令所完成的任务）。

2．虚拟机实例分析

为了理解虚拟机的运作方式，这里提供一个P-code编译的小程序VBP-code-Ex.exe来进行说明，其源码如下：

[image]

这个程序只处理Command_Click事件，同时用到了InputBox，以便

用rtcInputBox函数设断。

用OllyDbg加载这个程序，在命令行插件中输入bp rtcInputBox，按下F9键运行。然后单击OK按钮，中断如下：

[image]

按下“Ctrl + F9”键返回，当InputBox弹出以后任意输入一个整数，然后继续单步跟踪，直到下面的代码处：

[image]

[image]注意：由于msvbvm60.dll版本不同，读者电脑显示的代码或地址会与本书不同。

读者如果不明白这些指令的意图，就会迷失在行为相似的代码之中。事实上，这几条指令正是虚拟机读取P-code伪代码的引擎部分。在这里，尤其值得关注的是它读取了什么，而不是执行了什么。为了说明这些指令的功能，首先要了解VB P-code伪代码的格式：

[image]

这是一句典型的VB P-code指令，其助记符为LitVarStr，表示将一个Variant型的字符串入栈。3A是指令的操作码，0003是操作数，是以某种形式标记的字符串地址。现在有足够的信息来解释虚拟机所做的一切了，代码如下：

[image]

虚拟机并没有用简单的条件判断来识别操作码，而是采用了跳转地址表法，把执行流直接引导到相应的解释单元。6A37DA58h是地址跳转表的首地址，eax保存了下一条指令的操作码，由于每一个跳转地址是一个dword，所以用eax乘以4的值加上跳转表的基地址来索引下一条指令的解释单元。当然，由于每一条指令的长度是不同的，所以前面提到的读取P-code伪代码的引擎部分并不完全相同。明白了虚拟机的解释原理，跟踪P-code程序就方便多了，一旦走到读取P-code伪代码的引擎部分，就意味着虚拟机开始解释下一条指令了。尽管如此，调试P-code程序还是比调试本地机器码编译的普通可执行程序困难得多，因为在虚拟机的工作流程下无法随时回顾前面执行过的指令。

下面来看看加法在虚拟机中是如何被解释执行的。按F8键跟踪代码来到：

[image]

走到这里时al的值为FB，这是Variant型变量算术运算伪指令的操作码，esi则指向了该操作码后的一个次操作码94，代表加法运算。跟随跳转地址表来到：

[image]

这里是一个二级跳转表，注意6A37DE58这个值和前面的跳转地址表基址不同了，这说明VB P-code算术运算伪指令采用了二级跳转来解释执行。由于一个字节的操作码可以表示的操作指令种类最多为256个，为了解决指令数不够的问题，Microsoft对部分伪指令进行二级跳转解释执行。

[image]

很明显，Variant变量的算术逻辑运算伪操作都分布在这一区域，待调用函数__vbaVarAdd的地址被装入ebx，继续向下来到：

[image]

在OllyDbg朴素而强有力的逐行追击中，VB P-code虚拟机的解释机理尽可一览无余。

8.3.2　动态分析VB P-code程序

毋庸置疑，OllyDbg完全可以胜任VB P-code程序的调试，然而在大部分情况下，它并非是最好的选择。基于通常的调试习惯，人们总是希望能够直接跟踪可执行文件本身的每一条指令，以便直观地了解程序实现的功能。从这个角度来说，通用调试工具是分析VB P-code虚拟机的优秀选手，但是却无法将更多的注意力集中在P-code程序本身的功能上。

伴随着这样一种想法，功能强大的VB P-code专用调试器WKTVBDebugger应运而生了。这个具有划时代意义的VB P-code调试器，掀开了解释型语言程序调试策略的崭新一页，它不仅仅给VB P-code程序的调试带来了极大的方便，其编写思想和运作机制也给人以深刻启发。在本节中，笔者将为大家简要介绍这一款调试器的各种特性和使用方法。

1．介绍

WKTVBDebugger以单个伪代码指令为执行单元，包括跟踪、设断、修改等各个方面都完全以伪代码为基础，彻底屏蔽了虚拟机的解释细节，如同直接调试P-code可执行文件的功能。举例来说，当程序执行一条Variant型变量的加法伪指令时，读者在调试器中跟踪的将不再是冗长繁复的虚拟机解释代码，而是AddVar这一条指令（严格地说，AddVar只是助记符），不仅操作码如此，每一条伪指令的操作数也可以通过堆栈来方便地观察。

在基于本地机器码的调试器实现中，通过设置单步跟踪标志位，可以在每执行一条指令以后产生单步调试断点，从而将控制权转移给调试器，调试器在执行下一条指令前可以观察寄存器的状态并修改程序的执行流程；此外，调试器还可以通过在指定的指令位置插入CC（Int 3）设置断点，捕获异常并取得控制权。特别地，在Win32环境下，这类断点在Ring 3下形成异常，由Windows系统向调试器报告异常事件，调试器

由此获得处理异常的优先权。然而，对于由虚拟机解释执行的伪代码，因为系统并未给程序开发人员特意留出标准的调试接口，异常的处理对用户而言也是透明的，传统的调试器原理便无法照搬过来。

WKTVBDebugger的实现，与VB P-code虚拟机运作机制以及伪指令的研究是密不可分的。其作者Mr. Silver和Mr. Snow采用了一种类似于Hook的方法，把调试器代码插入到虚拟机和VB P-code伪代码之间，从而巧妙地解决了如何让调试器取得控制权的问题。观察一下VB P-code虚拟机伪代码读取引擎：

[image]

在这一模式中，寄存器esi始终指向伪指令流，虚拟机读取下一条伪指令的操作码以后，又把esi指向次级操作码（对于具有多级操作码的伪指令）或者操作数（对于具有单级操作码的伪指令），每一条jmp指令都通过跳转地址表移向下一条伪指令的解释单元……如果让这里的跳转地址指向自定义的调试代码，不就可以在下一条伪指令执行前取得控制权了吗？调试所需的初始化工作，仅仅是把6A37DA58所指向的跳转地址表中的所有项替换为调试循环代码的入口地址，一旦在此截获P-code指令，由于al保存着伪指令的操作码，而esi则指向伪指令的操作数，调试器便可以把对应的伪指令助记符在反编译窗口中显示出来，并判断当前的地址是否需要中断……无论如何，程序的执行流程已尽在掌握。一旦调试器的任务完成，就恢复现场的寄存器环境，根据原来的跳转地址表移至下一条指令的解释单元。这就是WKTVBDebugger Hook的基本原理。

调试器的基本框架有了，随之而来的是伪指令的解释问题。Microsoft定义了VB P-code指令规范，但却没有把它们公开。这不是一个大问题，无论如何，这套指令规范是可以获得的。这里要感谢Josephco，他的VB P-code反编译器Exdec在这方面做了许多先驱性的工作。另外，笔者推荐两款相当不错的国产VB P-code反编译器：万涛编写的VBExplorer和ljtt编写的VBParser。严格地说，VBExplorer并不是一个纯粹的反编译器，它还可以编辑修改VB控件的各种属性，但反编译功能做得还不精。相比之下，VBParser是一个专业的反编译器了。

2．安装

执行WKTVBDebugger文件就能完成安装。但还有几个方面要注意一下，如果不能正常装载程序，请如下尝试。

- 将目标软件复制到WKTVBDebugger安装目录里调试，即与Loader.exe同一目录；
- 将安装目录的WKTVBDE.dll文件复制到系统目录里；
- 将MSVBVM60.DLL替换成2003年以前版本。

3．使用

运行WKTVBDebugger后，打开目标程序，单击菜单“Action/Run”装载程序。输入用户名及序列号后，单击“确定”按钮就能中断在WKTVBDebugger里，如图8.4所示。

[image]

图8.4　WKTVBDebugger主窗口

主窗口由3个基本部分组成：代码（Disassembly）、日志（Log）以及堆栈（Stack）窗口。

（1）代码窗口

这个窗口显示当前执行程序的反汇编伪代码。指令格式如下：

[image]

其中“XXXXXXXX”是指令的内存虚拟地址，跟着的两个数字是以十六进制格式显示的机器码。注意，只有第一个机器码被显示了，后面的是P-code伪指令（会根据变量和自变量的不同而不同）。另外，在该窗口中单击鼠标右键将打开命令菜单。

（2）日志窗口

当程序执行时，显示程序所有变量的信息和提示消息。所以当跟踪调试时，注意这个窗口是很重要的。它将提供一些有价值的信息，最重要的是，它将显示当前指令执行时所进行的操作。

（3）堆栈窗口

堆栈窗口有几个模式：字节、字或双字。P-code运行有别于传统的CPU结构，传统的CPU执行依赖于寄存器和堆栈；而P-code只使用堆栈，所以堆栈窗口非常重要，各种指令都通过堆栈来交换数据。

4．机器码与助记命令

机器码与助记命令表（Opcode and Mnemonics Table）很重要，具体参考WKTVBDebugger的帮助文件。掌握VB P-code的关键就在这些助记命令上。这里只列出几个常用的助记命令。

• BranchF：机器码是1Ch，3个字节。条件跳转指令，如堆栈的值是0就跳转。单击“Analize BranchX”按钮可以了解当前进程中的所有条件跳转指令的位置。

• BranchT：机器码是1Dh，3个字节。条件跳转指令，如堆栈的值是FFFFFFFh（-1）就跳转。

• Branch：机器码是1Eh，3个字节。无条件转移。

• EqVarBool：机器码是33h，1个字节。比较指令，比较两个变量，根据结果将-1或0压进堆栈。

• LitI2_Byte：机器码是F4h，2个字节，将数据压入堆栈。

• ConcatStr：机器码是2Ah，1个字节。字符串连接指令（相当于C语言中的strcat函数）。此指令单步跟踪（Step Trace）时，会在日志窗口中留下相应字符串连接结果。所以，可以单击“Opcodes”在此伪指令处设置断点，以便了解某字符串值。

• FLdZeroAd/CVarStr：取字符串指令，特点同ConcatStr。

8.3.3　伪代码的综合分析

尽管Microsoft为VB P-code定义了一整套伪指令助记符，但是并没

有权威的技术文档解释这些伪指令执行的细节。既然WKTVBDebugger作为一个伪代码级的调试器已经屏蔽了VB P-code虚拟机的解释过程，为什么还要颇费周折地去了解这些伪指令执行的细节呢？在调试的过程中，读者一定会遇到这样的问题：假设用WKTVBDebugger步过了AddVar这一条伪指令，试问如何才能获得它的操作数和操作结果？既然VB P-code虚拟机是基于堆栈的，那么操作数和操作结果一定存放在堆栈中。实际情况诚然如是，然而它们究竟是以怎样的形式存在呢？是单个的操作数，是指针，还是其他复杂的数据结构？对于不同的P-code伪指令，其存放形式也是迥然相异的。如果在调试的过程中无法正确地获知和修改每条指令的操作数和操作结果，那么WKTVBDebugger的功能也就止步于静态反编译而已。

这里仍然以VBP-code-Ex.exe为例说明如何用现有的工具来解决上述困惑。首先使用ljtt的VBParser解析VB P-code.exe，得到伪代码如下：

[image]

以上就是Command_Click事件响应代码的开头部分，由于VB P-code虚拟机以流的形式顺序读入每一句伪指令，然后通过一个跳转地址表找到相应的解释代码，为了跟踪它解释伪指令的细节，就必须在伪指令的操作码上下内存访问断点。第一句伪指令LitVar_Missing从004019C4（虚拟地址）开始，用OllyDbg加载VBP-code-Ex.exe，在转储窗口中来到004019C4，对第一个字节（操作码）下内存访问断点，按F9键执行，单击OK按钮，中断在这里：

[image]

按F8键往下走一步，来到这条伪指令的解释单元：

[image]

看一下这些指令执行完以后的堆栈：

[image]

现在这个伪指令的动作明确了，LitVar_Missing执行时，把一个虚拟地址压入堆栈，这个虚拟地址指向0000000A，00000000，80020004。实际上，这句伪指令的功能就是在堆栈中提供一个空参数，其堆栈完全没有参考价值。在下面的说明中笔者将省略对虚拟机伪代码读取引擎的注释，因为这部分都是一样的。

现在来看看4019D3处的伪指令LitVarStr。老规矩，在4019D3处设内存访问断点，按F9键中断在下面的地方：

[image]

跟随跳转来到：

[image]

同样地，有必要观察一下堆栈：

[image]

结合上述跟踪分析，LitVarStr伪指令对操作数的获取方法就清楚了：首先在转储窗口观察12F544处的内容，向后移动8个字节，得到虚拟地址401434，观察401434处的内容，就是入栈的字符串参数了。

相应地，笔者在WKTVBDebugger中演示一下操作的过程：

① WKTVBDebugger加载VB P-code.exe；

② 在Form Manager中对Command1控件设断点：

③ 单击OK按钮；

④ WKTVBDebugger中断在下面的地方：

[image]

⑤ 注意注释标出的这条伪指令，单步走过这条指令时，右上角堆栈窗口显示如下（为了便于观察，在右侧的单选框中选择DWORD）：

[image]

⑥ 按下“Ctrl + M”键打开转储窗口，在Address to Dump组合框中输入0012F524，得到：

[image]

⑦ 记下这个指针，输入到Address to Dump组合框，现在这个Unicode字符串终于露出了真面目：

[image]

当然，就该指令本身而言，WKTVBDebugger已经在日志窗口中输出了其操作数，所以要观察这个字符串大可不必那么麻烦，但是对于其他没有日志记录的伪指令，这套分析方法仍是具有启发性的。

8.3.4　VB P-code攻击实战

本节以CyberBlade编写的一个中等难度的VB P-code CrackMe作为范例来初步介绍VB P-code程序的调试过程，以帮助读者熟悉一些常见的P-code伪指令。

由于反编译器往往也提供了一些有用的信息，因此首先以Josephco的Exdec来生成该CrackMe的反编译代码，对照WKTVBDebugger的代码窗口进行调试。在WKTVBDebugger的Form Manager中对Check按钮下断点，填写用户名“cyclotron”和序列号“131421”，单击“Check”按钮以后，首先是关于用户名和序列号的存在性校验。

[image]

接着是关于序列号长度合法性的校验：

[image]

从下面开始的两个循环是这个CrackMe的算法核心，其中所涉及的指令和函数对P-code程序的调试都具有相当的指导意义。

第一个循环（For结构）：

[image]

[image]

总结这个循环的算法如下：

（1）逐位读取用户名字符：

“cyclotron”[image]‘c’，‘y’，‘c’，‘l’，‘o’，‘t’，‘r’，‘o’，‘n’

（2）分别转换为十进制的数字字符串：

99，121，99，108，111，116，114，111，110

（3）顺序连接所有的数字字符串：

“9912199108111116114111110”，记为S1

注意：这个字符串是以Unicode编码的形式在内存中出现的。

下面来看第二个循环（While结构）：

[image]

在这个位置WKTVBDebugger会遇到一点小小的困难，它无法现场输出这个浮点立即数的标准形式。如何获得这个浮点数呢？可以利用OllyDbg的数据窗口以标准形式输出浮点数。

[image]

上述循环运算过程可以总结如下：

（1）变换S1到9位以下的十进制形式：

[image]

（2）变换S1到S2：

S2 =（S1 xor 30F85678h）- 0D8B3h = 667574632

接下来进入另一个循环：

[image]

[image]

令人意外的是，这个循环对程序的运行路线没有任何影响，鉴定为作者布下的迷魂阵。尽管如此，最后的验证也已经不远了。

[image]

[image]

这部分的算法总结：
strlen（用户名）= 序列号 - S2
据此得到正确的注册信息：
注册码 = strlen（"cyclotron"）+ 667574632 = 667574641

第9章　.Net平台加解密②

本章将简要介绍微软.Net框架下的安全问题，内容基本涵盖.Net下本机Windows程序保护的各个方面，包括强名称、混淆、加壳、加密等常用保护手段，以及相应的逆向方法。

9.1　.Net概述

2002年微软正式发布了.Net的第一个版本，从此，基于.Net平台的各种软件产品逐渐增多，最新版本的Windows Vista已经直接内置了.Net Framework 3.0。由于越来越多的企业及程序开发者将产品定位在该平台上，.Net软件的保护就成为一个不可回避的课题摆在大家面前。本节主要介绍基本概念，读者应将重点放在理解.Net框架的概念和程序的运行方式上。

9.1.1　什么是.Net

.Net是微软设计的独立于操作系统之上的平台，可以将它看成一套虚拟机，无论机器运行的是什么操作系统，只要该系统安装了.Net框架，便可以运行.Net可执行程序，享受基于.Net的各类服务。上句话是从用户角度出发的观点，如果从Windows系统的角度来理解，.Net就是一系列运行于Ring 3层的DLL文件。

对于加解密的学习者，可以从以下三个方面来理解.Net。

（1）统一了编程语言：无论程序是用C#，还是C＋＋，或是VB编写，最终都被编译为.Net中间语言IL；

（2）扩展了PE文件的格式：可执行文件中不再保存机器码，而是IL指令和元数据，部分结构也被改变用于保存.Net的相关信息；

（3）改变了程序的运行方式：Windows不再直接负责程序的运行，而由.Net框架进行管理，框架中的JIT引擎负责在运行时将IL代码即时编译为本地汇编代码再执行。

到本文写作时为止，.Net的正式版包括1.0、1.1、2.0、3.0和3.5。不同版本的.Net框架可以在同一个系统中共存，这是.Net相对传统DLL式程序兼容性的重要改进之一。本章内容以目前最流行的Windows XP加.Net 2.0平台为主，兼顾1.1平台。2.0版之后的.Net内核基本相同，只是新增了一些企业功能。

9.1.2　几个基本概念

学习新平台，总要接触一些新名词，本节就介绍几个与.Net平台加解密关系最为密切的基本概念。如果读完本节仍不是很清楚这些概念的含义，也没有关系，读者将在后面的实践中一步步掌握它们的本质。

MSIL：微软中间语言（Microsoft Intermediate Language），大多数时候简称IL。.Net下有很多种高级语言，常见的包括C#、C＋＋/CLI、VB.Net，但无论哪一种语言，最终在编译后都生成IL。IL是.Net唯一能读懂的语言，也是唯一可执行的语言。大多数时候，对.Net程序进

行分析和调试，就是对IL语言进行分析和跟踪。由于运行完全受.Net监控，因此IL属于托管（Managed）代码，与之对应的是本机代码如X86/64汇编，被称为非托管（Unmanaged）代码。如同Win32下要掌握汇编一样，.Net下必须掌握IL。

CLR：CLR是Common Language Runtime的简称，中文名叫通用语言运行时。CLR是.Net框架的核心内容之一，可以把它看为一套标准资源，理论上可以被任何.Net程序使用。它包括：面向对象的编程模型、安全模型、类型系统（CTS）、所有.Net基类、程序执行及代码管理等。CLR是IL语言运行的环境，就像Windows是普通PE程序的运行环境一样。在Windows中，整个CLR系统的实现其实就是几个Ring 3层的DLL，比如mscorwks.dll、mscorjit.dll，它们共同的特点是前缀均为mscor。

Metadata与token：在.Net中，元数据（Metadata）描述了一个可执行文件的所有信息，包括版本、类型的各个成员（方法、字段、属性、事件）等。一个文件要成为有效的.Net可执行程序，必须包含正确的元数据定义。由于元数据将所有的程序信息保存在文件中，并很容易被相应的反编译工具读取，所以，对元数据的加密是现有加密软件的重点之一。为了区分各项元数据，需要提供一个单独的标识，这就是token，它是同一程序中区分和定位不同元数据的依据。读者将在“PE文件结构扩展”一节中详细了解什么是元数据。

JIT：即时编译（Just In-Time Compile），这是.Net运行可执行程序的基本方式，也就是在需要运行的时候，将对应的IL代码编译为本机指令。传入JIT中的是IL代码，导出的是本机代码，所以部分加密软件通过挂钩JIT来进行IL加密，同时又保证程序正常运行。同解释执行的代码相比，JIT的执行效率要高很多。

Assembly和Module：程序集和模块，它们是构成.Net程序的基本元素。两者之间是包含的概念：一个或多个含有可执行代码的Module，加上一些必要的控制信息，构成了一个Assembly。因此，程序集是.Net中可执行程序的基本单元，通常遇到的可执行文件就是一个程序集，包括.exe和.dll。有时会出现以.netmodule结尾的文件，其中也包含可执行代码，但不含清单（Manifest）信息，因此只是一个模块，而不能单独成为Assembly。

Type和Method：类型和方法，这是面向对象程序设计中的概念。类型是.Net程序构成的基本元素，最常见的类型是类（class），其次还有结构（struct）和枚举（enum）。类型可以有很多成员（member），最重要的成员是方法（method）。方法是代码的基本单元，可以看作面向过程中编程语言中的函数，所有的代码都被定义在某个类型的某个方法中，定位关键方法是.Net解密中的重要步骤之一。

AppDomain：应用程序域，这是.Net独有的概念。.Net中的进程和线程与Win32平台下的不同，.Net中程序运行的基本单位是Assembly，几个Assembly可以构成一个AppDomain，通常代码只能访问本域中的数据，几个域可以运行在一个传统意义的进程下。AppDomain实现的功能有点类似进程，区分方法也很简单，前者是CLR中引入的概念，后者是操作系统（如Windows）的概念。

9.1.3 第一个.Net程序

了解一个新平台的最好方法是亲自写一个小程序，下面就请读者自己动手编写第一个.Net程序。在这之前，确定已经下载并安装了微软.Net Framework SDK，1.1和2.0版均可。C#代码如下：

[image]

代码中建立了一个类class1（因为.Net是面向对象的平台，所有的代码应定义在某个类中），且建立了一个公共的（public）静态（static）方法Main作为入口点，名称上和Win32下相似。首行加入了using语句来表示本程序引用了System空间的类和方法，所有的.Net程序都会引用这个名称空间。

新建一个文本文件，输入上面的代码并保存为.cs（C#源码的默认扩展名），然后在SDK的命令行中键入：csc hello.cs，回车执行，一个.Net程序便生成了。（除非另外说明，本文所有的代码均在.Net 2.0下编译通过，工具为Visual Studio.net 2005和SDK自带的编译器。）用SDK自带的反汇编工具ildasm.exe看一下它的IL代码。

[image]

这便是C#编译器生成的IL代码。看一下代码流程：ldstr读入字符串，call调用系统方法WriteLine将字符串输出到屏幕，再调用ReadLine使程序暂停，最后是ret返回指令，中间还有两个空指令nop。可以看出，同asm相比，IL语言既有相同的指令助记符（如call、nop），又有新指令（如ldstr）。严格地说，IL是一种高级语言，它支持面向对象，且不直接操作内存地址，它的语言特征反映了.Net框架的实现原理。

9.2 MSIL与元数据

MSIL与元数据是相辅相成的两个概念：IL语言对元数据进行操作，而其本身又为元数据所定义，两者共同构成了.Net程序的基本要素。熟练地掌握IL与元数据是进入.Net加解密领域的敲门砖。由于两者同时被存储在PE文件中，且PE文件又反映了一个系统的基本运行框架，因此下文首先介绍.Net平台下PE文件结构的扩展，其中与Win32下重复的内容不再赘述。

本节的重点是理解MSIL汇编、元数据与PE结构三者间的关系，读者可结合文件结构信息查看工具，边实践边理解本节内容。

9.2.1 PE结构的扩展

回顾Win32平台下的PE结构（参考第10章），其中有几个Data Directory是未被使用的，.Net对其进行了扩展，其中第15项数据目录便指向了Common Language Runtime header。以9.1.3节的hello.exe程序为例，在文件偏移168h处可以找到表示CLR头的数据目录，如图9.1所示，即RVA = 2008h，Size = 48h。（考虑到大多数读者已经能熟练使用工具查看PE结构，因此本小节在介绍结构的过程中，大多直接以十六进

制数据为例，便于读者对比分析。）

[image]

图9.1　第15项数据目录指向CLR头

可以发现，这个偏移指向了exe文件的.text区块。.text区块是.Net对PE结构改变较多的地方，传统Win32平台下该节通常只存储asm汇编指令，而.Net中所有的元数据和IL代码均存储在.text区块中，图9.2所示是.text区块的大致结构。

[image]

图9.2　.Net程序.text区块的构成

.text区块中的第二项便是CLR头（又可称CLI头），该头结构的定义在SDK中的CorHdr.h里可以找到，名为IMAGE_COR20_HEADER（无论是Windows还是.Net，SDK中include目录里的一些头文件往往是发掘系统隐藏信息的好地方）。精简后的头结构代码如下：

[image]

IMAGE_COR20_HEADER结构的各项说明如表9-1所示。

表9-1　Common Language Header结构与说明

[image]

其中Flags项定义了该exe文件的最基本性质，包含以下设置：

[image]

图9.3所示为hello.exe的CLR头数据，请读者自行与表9-1中各项进行对照。

[image]

图9.3　hello.exe的CLR头数据

根据表中最重要的MetaData项，来查看元数据在PE文件中的存储格式。在示例程序中，元数据的RVA和大小分别是206Ch和290h，转换为文件偏移就是26Ch处开始。注意到这个地址与刚才CLR头的结束地址24Fh之前还有一点空隙，IL代码正是存储在这段空间里。

MetaData结构以一个元数据头开始，代表元数据的定义由此开始。看一下官方对MetaData Root的定义：

[image]

表9-2给出了元数据头中各项的意义。

表9-2　MetaData Root结构与说明

[image]

紧接着元数据头便是几个流数据的头，流按存储结构的不同分为堆（heap）和表（table），上述头信息指明了这些堆和表的位置及大小。.Net中共有以下几种流，但不是每个文件中都包含所有的流。读者可以先学习#~、#Strings和#US流，因为这三种流几乎在每个.Net程序中都会出现，并且和加解密的关系最为密切。随着学习的深入，再掌握#Blob流的结构。

- #Strings

UTF8格式的字符串堆，包含各种元数据的名称（比如类名、方法名、成员名、参数名等）。流的首部总有一个0作为空字符串，各字符串以0表示结尾。CLR中这些名称的最大长度是1024。

- #Blob

二进制数据堆，存储程序中的非字符串信息，比如常量值、方法的signature、PublicKey等。每个数据的长度由该数据的前1～3位决定：0表示长度1字节，10表示长度2字节，110表示长度4字节。

- #GUID

存储所有的全局唯一标识（Global Unique Identifier）。

- #US

以Unicode格式存放的IL代码中使用的用户字符串（User String），比如ldstr调用的字符串。

- #~

元数据表流，最重要的流，几乎所有的元数据信息都以表的形式保存于此。每个.Net程序都必须包含。

- #-

#~的未压缩（或称为未优化）存储，不常见。

每种流都具有共同的头结构，如表9-3所示。

表9-3　Stream header结构与说明

[image]

相应的C＋＋代码如下：

[image]

示例hello.exe中共含5个流，没有“#-”流。以“#~”为例，从图9.4中看出，它的偏移是6Ch，大小是E8h，也就是开始地址为26Ch＋6Ch＝2D8h，正好位于最后一个流“#Blob”与4对齐后的位置。也就是说，紧跟着Stream头定义的便是“#~”流的内容。由于“#~”流是最重要的元数据存储区域，因此下文仅就该流的结构继续深入，而其余各流的结构则由读者自己查阅资料进行学习。

[image]

图9.4　Stream头的数据

示例中2D8h处指向的是“#～”流，也就是元数据表流，该处数据按表9-4中的结构进行组织，其中的内容包括了元数据中有哪些表，各个表的性质等。

表9-4　Metadata Table Stream结构与说明

[image]

同样，该结构也有相应的C＋＋定义：

[image]

由于Valid项长度为8字节，因此很容易得出.Net中最多可定义8×8＝64个表，而实际上.Net已定义的只有45个表，见表9-5。判断Valid被置1的二进制位，便可以得出该程序中使用了哪些表。以hello.exe为例，由于int64在内存中以反字节顺序存储，可得valid＝0000000900001447h，相对应的为Module，TypeRef，TypeDef，MethodDef，MemberRef，CustomAttribute，Assembly和AssemblyRef这8个表。

表9-5　元数据中所有的表（斜体为.Net 2.0新增的）

[image]

紧接着Metadata Table Stream头的是一串4字节数组，每个双字代表该表中有多少项记录（record），8个表共32个字节。然后便开始各表的数据了，排在第一位的自然是Module表。44个表各有各的结构，为节省篇幅，在这里只介绍比较典型的Assembly和MethodDef表，其余表结构请读者自行查阅资料。

Assembly表主要定义了该程序集的基本性质，其结构见表9-6，注意到表中最后三项索引值为2（4），表示2或4字节，这正是由Metadata Table Stream头中的Heaps项决定的，由于示例程序中该项为0，因此所有的索引值大小均为2字节。

表9-6　Assembly Table结构与说明

[image]

MethodDef是另一个很重要也很有趣的表，因为它不但指出了该方法IL代码的位置，还限定了方法的属性，以及该方法如何被调用。MethodDef结构见表9-7。

表9-7　MethodDef Table结构与说明

[image]

如果读者是第一次接触元数据和.Net的扩展PE结构，则最好使用十六进制编辑器，与表结构一项项对照。而在平时的分析和逆向过程中，有多种工具提供了直接浏览PE结构和元数据的功能，如Spices.Net、Dotnet Explorer、Researcher.NET和CFF Explorer等。一般的元数据查询与修改使用CFF Explorer比较方便，图9.5中显示了用该工具读入hello.exe后以树状结构显示的元数据。

[image]

图9.5　用CFF Explorer查看.Net PE的元数据

关于元数据最权威的参考，莫过于ilasm的作者写的《Inside Microsoft .NET IL Assembler》，对应于2.0平台的第二版名叫《Expert .NET 2.0 IL Assembler》，本节未详述内容均可在该书中找到，请读者自行参考。

9.2.2　.Net下的汇编MSIL

.Net平台下的程序，无论开发时使用哪种高级语言，最终都被编译为微软中间语言MSIL。IL与上述几种高级语言相比，显得更加底层，因此又被称为IL汇编。而实际上IL与asm汇编相比，又算得上高级语言，最明显的特征莫过于IL不直接和内存地址打交道。在.Net平台刚出现时，各种加密手段还不成熟，所以.Net下的破解基本上就是对IL反汇编代码进行阅读，随后写出注册机或直接在IL代码中“爆破”。现在，.Net下的加密手段越来越复杂，但IL代码仍是最重要的突破口之一。对于加解密来说，IL就像Win32下的asm，不是可学可不学，而是必须掌握的知识。（C＋＋/CLI混合编译的程序，其中既可保存托管代码，又可保存本地代码，不属于本章讨论范围。）

先来看一个例子，新建一个文本文档，键入以下代码后，保存为.il文件。在SDK的命令行里进行编译：ilasm sample922.il，最终会生成一个可执行文件。

[image]

[image]

该段代码的功能是分两行输出“1 + 10 = 11”这个字符串。寥寥20多行代码，包含了IL语言的最基本要素：

（1）IL源文件扩展名为.il。

（2）IL源文件中，用“//”号表示行注释，还可以用“/*　*/”表示注释块。

（3）一个.exe文件必须有入口。IL源文件中，入口方法名不一定要求为Main（还记得C#吗），而用.entrypoint表示。

（4）.assembly定义本程序集，而.assembly extern则定义被引用的程序集，两者分别对应元数据表中的Assembly与AssemblyRef。mscorlib是所有.Net程序的基础，每个程序都会引用它。

（5）所有的代码必须定义在某个类的某个方法中，代码中

addTwoInts方法就定义在class1类中。

（6）对于本地变量（由.locals定义），可以采用名称引用，也可以用序号表示。代码中只有一个本地变量val，因此在取val的值时，既可以用ldloc val，也可以用ldloc.0。

（7）读入常数值时，对于0～8，可以直接用简短指令，形如ldc.i4.1；而对大于8值的数值，如程序中的10，则必须用完整指令，如ldc.i4 10。

（8）调用某个方法时，必须完整地写出方法的返回值、空间名、类名，最后才是方法名，以及方法的参数。

（9）IL中也有空指令nop，不过它的十六进制编码是00h，而不是Win32 asm中的90h。

IL语言最大的特点是以堆栈为基础进行操作，通常不直接操作寄存器和内存，因此学习难度相比Win32asm要低。下面模拟示例代码的执行（见图9.6），来解释什么叫以堆栈为基础操作。示例图中小箭头指向当前栈顶，堆栈的生长方向由下往上，堆栈的名称叫evaluation stack，这是.Net内核给出的逻辑概念。

[image]

图9.6　IL代码操作堆栈流程示意图

图9.6给出了两个信息：一是有效的IL程序，必须保证堆栈的平衡，在方法开始时堆栈若为空，则在方法结束时堆栈也必须为空，这一点与Win32下的堆栈平衡有点类似；二是部分IL指令直接对堆栈进行操作，比如在堆栈中取操作数，或是将计算结果压入堆栈。其中第二点与Win32下asm指令相差比较大。以跳转指令为例，asm中的jz指令判断的是CPU的Zero标志位，堆栈数据对它没有影响；而IL中除了br和br.s外（直接跳转），其他以字母b打头的条件跳转均要求将栈顶的一个或两个元素取出并进行比较，指令结束后的堆栈已经被修改。所以，喜欢“爆破”.Net程序的读者应该注意，在进行跳转指令patch的时候，一定要注意堆栈的平衡。

解释完IL的堆栈操作原理，这门语言就已经学了一半，剩下一半是记忆各类指令助记符、关键字和代码格式。这里不再将所有的指令列出，而是在后文的分析过程中逐步介绍出现的指令及其具体功能。表9-8中列出了绝大部分IL指令所用的英文名称缩写，上半部分为操作数简称，下半部分为操作符简称，读者只需记住这些基本的英文单词，就可以对所有IL指令做到望文生义。

表9-8　IL代码英文缩写意义

[image]

很少会有开发人员直接使用IL编程，但对于研究.Net底层的解密爱好者来说，使用IL编程的机会相对较多，比如直接利用ildasm反编译源程序，并使用其中的代码编写注册机。由于ilasm编译器提供的纠错功能很弱，因此就算程序顺利编译，也可能在运行时报错。有三种方法可避

免此问题，一是尽量正确地使用IL，不让源代码中出现错误或少出错误；二是利用SDK中的辅助工具Peverify.exe，它不但可以验证IL代码的正确性，还可以验证程序元数据的有效性；三是利用第三方工具，比如Spices.net中的Pe Verify功能。

9.2.3　MSIL与元数据的结合

IL以元数据为操作对象，同时本身的执行又受到元数据的限定，两者的关系密不可分。元数据在IL中通过token引用和定位，token是元数据项的唯一标识。下面，用ildasm对9.1.3节的hello示例程序进行解码，看一下元数据在IL中的表示。注意将View菜单中的“Show bytes”和“Show token values”两项选中，以便直接观察指令的十六进制数据和各个token值。

[image]

从上面的代码看出，token值实际上就是一个uint32数值AABBBBBBh，其中前一个字节AA指出了它对应的表，后三个字节BBBBBB指出了它在表中的位置，也就是记录索引（RID, Record Index）。将本段代码出现的所有token（被包含在“/*　*/”中）列出表来对比一下就很清楚了，如表9-9所示。

表**9-9**　代码中出现的**token**值及其对应的表

[image]

细心的读者会注意到，这里还有一个token没有提到，就是70000001h，该值比较特殊，因为70h没有任何对应的表。这里只要记住，以70h开头的token对应的都是用户字符串，后三个字节000001h对应该字符串在#US流中的偏移（注意这里不是索引）。用户字符串流中偏移1处便是“hello,.net fans!”，这便是ldstr指令读入的值。这也说明了IL为什么是一种高级语言，因为它不是直接和内存地址打交道，而是采用token的方式（还记得Win32编程中的指针吗，那些指针都直接代表内存地址）。除上述已经出现的表外，其余表均按前文所列的顺序进行编码，比如1Bh开头的token值对应十进制27位的表TypeSpec。值得注意的是，45个表中只有23个表可以用token表示，剩余22个只用于内部使用，在MSIL中是不可使用的。

介绍完token的概念，下面再介绍一下什么是签名（signature）。回顾上面代码中的方法名下方有这样一行注释：

[image]

这行注释表示了Main方法的signature是00 00 01h。从PE结构上看，signature就是存储在#Blob中的一段二进制数据，它的作用是描述特定元数据的性质。.Net中共有6种表引用了signature，分别是Field、MethodDef、Property、MemberRef、StandAlongSig和TypeSpec。代码中关于Main方法的//SIG: 00 00 01按如下方法解码：

第一个00：任何signature的第一个字节都代表calling conventions，它定义了该sig的类型，是method、field或是property。00h代表IMAGE_CEE_CS_CALLCONV_DEFAULT，意思就是普通（默认）的方法，含定长的参数列表。

第二个00：代表方法中的参数个数，Main方法没有参数，因此为0。

第三个01：代表方法的返回值类型，其中ELEMENT_TYPE_VOID = 01h。

这样，一个method便被token和sig完全确定了：方法的代码根据token在相应表中查找，而方法的调用方式、参数个数及返回值类型被其相应的signature所限定。所有的sig数据保存在#Blob流中，在通常面对一般的保护时，并不需要用到sig解码，但一旦深入.Net核心（比如进行脱壳后的文件修复时），就会遇到自行解码sig的情况了。其他关于sig解码的详细帮助，请参阅SDK中Tool Develop Guide文档。

[image]注意：.Net中还有一种RID叫Record Identifier，记录标识，该标识仅用于.Net内部。若无特殊说明，本章中所用RID均指Record Index，即记录索引。

9.3 代码分析技术

到这里，读者已经掌握了分析.Net程序必需的基础知识，下面具体介绍几种.Net程序代码分析技术。与Win32类似，.Net下的代码分析也可分为静态和动态两种，本节就这两种方法分别加以介绍，目的是让读者掌握两种方法的原理及相应工具的使用，并能融会贯通。本节的示例程序是一个未加任何保护的CrackMe。

9.3.1 静态分析

静态分析就是用反编译工具将程序的指令字节反编译成IL指令或高级语言，通过阅读反编译代码掌握程序的流程与功能。由于.Net下的可执行文件同时保存有元数据与IL代码，其静态反编译出的代码具有极强的可读性，几乎等同于源代码。

.Net平台的反编译工具较多，其中最基础也最强大的便是.Net框架SDK自带的ildasm。毕竟是微软自己的产品，反编译出的代码也是最权威的。更重要的是，由ildasm得到的IL代码，经过修改，便可以用ilasm.exe再编译成可执行程序，这是.Net下patch的常用方式之一，微软官方对这一过程的命名叫round-tripping。

但最好用的反编译工具并不是ildasm，而是Reflector，因为后者可以将反编译出的IL代码转换为高级语言，比如C#、VB.Net等，高级语言的可读性实在比IL代码强很多。因此，下文主要介绍Reflector的使用。

运行Reflector，载入crackme.exe后，左边是以树状结构显示的文件结构及当前程序域中的所有Assembly，而双击某节点则会在右边显示该项的反编译代码，上方的下拉框用以选择代码的格式。为什么这里载入的是crackme.exe，但节点显示程序的名称却是WindowsApplication1？还记得在MSIL的章节中介绍的.assembly关键字吗，这里的名字正是由关键字定义的程序集名称，而与文件名无关。

看一下程序入口点。在WindowsApplication1上单击右键，选

择“Go to Entry Point”，直接来到程序的入口，这里是Main方法。双击Main节点，会在右方显示它的反编译代码，切换为C#（如图9.7所示）。

[image]

图9.7　Reflector的主界面

静态分析的主要任务是定位关键代码，分析程序流程。怎样在静态分析中定位计算注册码的关键方法呢？对于小程序，比如这个CrackMe，可以在树状结构中浏览并一一查看，对于拥有成千上万个方法的大程序来说，这几乎是不可能完成的任务。这时，需要使用Reflector的查找功能，查找的目标包括含敏感名称的方法、类型或者用户字符串。哪些字符串含敏感信息呢？如“activate”、“register”、“user name”、“password”和“crypt”等，这些字符串都有可能成为定位关键代码的突破口。示例中的CrackMe在注册成功时会显“Congratulaions”，因此就将它作为目标进行查找。（程序员也应该注意，尽量不要在程序中直接使用未经加密的字符串！）

在主界面上按F3键，进入查找界面，输入“congratulations”，并将查找类型切换为String（搜索字符串），搜索结果会告诉我们哪些方法调用了这个字符串。结果显示只有一个方法，就是crypt1。双击该结果，左边树控件中会将crypt1节点高亮显示，再次双击则会反编译出它的代码，如图9.8所示。

[image]

图9.8　Reflector的搜索功能

Reflector另一项很有用的功能是分析（Analyse），即分析被哪些项调用，而自身又调用了哪些项。接着上面的搜索结果，来到crypt1方法处，在节点上单击右键，选择“Analyse”，右边便显示出分析结果，其中显示出buttonTry的方法中调用了crypt1。在该项上单击右键，选择“Go to member”，来到该方法处，双击后便显示出buttonTry的详细代码。Analyse功能还可用在系统节点上，比如某程序采用了网络验证，便可以在WebRequest上进行分析，查看程序中哪些方法调用了网络功能。

Reflector还集成了许多插件，用以扩充它的功能，甚至包括了调试插件，这些都可以在作者的主页上（http://www.aisto.com）下载。Reflector如此强大和普及，使它成为了最容易被Anti的反编译软件，后文也将演示一些Anti方法。在这种情况下，就需要使用其他的工具来辅助静态分析。除Reflector之外，反编译工具还包括9rays Spices.net、Dis#、Decompiler.net、Xenocode Fox，这些软件都集成了除反编译外的其他许多功能，比如Metadata查看、反名称混淆功能等。

关于ildasm，由于它是最基本的反编译软件，因此众多混淆软件提供了Anti的功能，比如Spices.net混淆器。看雪论坛上已经有文章探讨过该原理，在2.0版的.Net中，只要代码中设置了

System.Runtime.CompilerServices.SuppressIldasmAttribute属性，便会造成ildasm拒绝反编译。

一般说来，定位出关键代码（注册码比较、激活、验证）的位置，程序就分析了一半。但通常情况是算法较复杂，特别是现在的程序几乎都经过了流程混淆，想直接静态分析出整个计算流程很难。这时，便需运用动态调试。

9.3.2　动态调试

运用调试工具对程序进行调试，便可以跟踪运算过程，实时观测指令的运算结果，若是注册码完整地出现在内存中，也可以在调试工具中直接显示。

结合不同的工具，.Net下调试有几种方法，第一种方法是用ildasm将程序反编译为.il文件，再用ilasm.exe带上/DEBUG信息编译回可执行程序，最后用SDK中自带的GuiDbg进行调试；第二种方法是用WinDbg配合.Net调试扩展SOS，最大的优势是可以直接利用微软的各种符号资源，但WinDbg的使用较复杂，不便于新手入门（WinDbg在.Net内核调试中显示出强大的功能，读者可自行测试）；第三种方法是使用直接调试的工具，如PEBrowseDbg和OllyDbg等。这里介绍使用PEBrowseDbg的方法，它的最大特点，一是方便，直接打开程序便开始调试；二是支持inter-op调试，IL与asm代码同时显示；三是断点功能强大。

先来看看PEBrowseDbg的基本功能。PEBrowseDbg提供了表9-10所示的断点类型。

表9-10　PEBrowseDbg支持的断点类型

[image]

用PEBrowseDbg对刚才的CrackMe进行调试，调试时的主界面如图9.9所示。直接载入CrackMe并运行，程序会中断在ntdll!LdrInitializeThunk处，单击继续，程序便中断在CrackMe的入口处。此时，左边树状显示区会显示程序域中的所有模块，展开CrackMe节点，会看到.Net下特有的两个节点，一是.NET MetaData，二是.NET Methods。而代码窗口中同时显示出了入口方法的IL与asm代码，深色的一行为当前EIP指向，标题栏中则显示当前中断位置。

[image]

图9.9　PEBrowseDbg调试主界面

在代码窗口中单击右键，会显示弹出菜单并提示更多功能，如Run to Selection，运行到所选中的指令处；Set EIP Here，设置下一条运行指令；Include MSIL，是否同屏幕显示MSIL代码和asm代码。很多选项在OllyDbg中也能够见到，这正是因为.Net下调试的实质是对JIT生成的asm代码进行的，其本质和OllyDbg调试Win32下的程序相同。

将CrackMe的.NET Methods展开，找到crypt1节点，并在该方法上下断点。对于这种简单的程序，可以直接通过浏览节点找到对应方法，

但对于大型程序则一般通过查找的方法定位某节点，方法是直接在节点的右键菜单中选择“Search From”。

下断点有两种方法，一是对某方法单独下断，二是对某个类型的所有方法下断，图9.10分别演示这两种方法的不同操作方式。

[image]

图9.10　两种下断点的方式（左：单独下断；右：全部下断）

既然在静态分析的过程中已经知道crypt1的功能是注册码验证，便可直接在该方法上下单个断点。按F5键继续运行后，会显示CrackMe的界面，在用户名和注册码框中随意输入数据，单击try按钮，便会中断在crypt1的入口处。运行到字符串比较的指令System.String :: op_Equality()处，对应的asm代码为call0x7934B8F0。

[image]

这时打开寄存器窗口（“View/Registers”），双击ecx，便会显示ecx所指内存的数据。可以发现它正指向了正确的注册码“dzEbDQYDAQA = ”。这样，便绕过了复杂的注册码计算过程，直接得到了答案，如图9.11所示。

[image]

图9.11　查看ecx所指内存中的注册码

由于注册码的明码完整地出现在内存中，所以可以直接查看。但对于没有在内存中完整出现注册码的程序，动态调试更多地是帮助分析运算过程。开发者也应该尽量避免在代码中直接进行完整注册码明码比较。

在Win32下，user32.dll导出的API函数MessageBox是常用断点之一，同样，在.Net下也可以直接对系统方法下断。以MessageBox为例，.Net中MessageBox的定义出自System.Windows.Forms.MessageBox.Show，在Forms.dll中查找MessageBox类，然后在Show方法上下断。图9.12显示出该类中有十几个Show方法（面向对象程序设计的重载机制），这时便可以用Add Breakpoints To All，给所有的方法下断，无论程序最终调用哪个，都可以被捕获。

[image]

图9.12　给系统方法MessageBox.Show下断点

PEBrowseDbg还提供了很多的设置功能，不过默认的就已经足够正常使用了。程序调试是很有意思的，三言两语无法涉及所有方面，更多的经验和技巧需要读者在调试过程中自己去总结归纳。

9.3.3　代码修改

.Net中PE文件的patch有三种方法：一是用ildasm将代码反编译为IL文件后，修改IL代码，再用ilasm编译回可执行文件；二是直接在PE文件中查找到IL代码对应的数据，在十六进制编辑工具中修改；三是利用工具将反编译代码导出为C#工程。下面分别以前两种方法为例介绍如何给CrackMe打补丁，而第三种方法可用到的场合实在是太少了。

[image]

图9.13　用ildasm反编译可执行文件

用ildasm反编译CrackMe，如图9.13所示，命名生成文件为1.il，会在相应目录下生成4个文件：1.il源代码文件，1.res资源文件，两个以resources为扩展名的.Net资源文件。用编辑器打开1.il，定位到如下代码：

[image]

代码的意图很明显：如果判断注册码失败，则brfalse.s跳转，成功则继续执行，并显示MessageBox。修改的方法很简单，不让brfalse跳转，注意前面提到的堆栈平衡原则，所以这里修改为pop而不是nop。

[image]

在SDK命令行中进行编译，注意包含资源文件，回车后生成了1.exe。

[image]

双击运行后，无论在用户名与密码框中输入什么内容，程序直接显示“注册成功”！

第二种方法，直接在PE文件中修改。怎么定位相应的IL代码呢？需要查找IL代码的十六进制字节。在ildasm的View菜单中打开“Show bytes”选项，双击某个方法，会显示该方法每条IL代码的字节。判断注册的关键代码如下：

[image]

可以查找2C 10 72 88 02 00 70，查找数据的多少以能唯一定位为标准。这里要注意intel字节反转顺序，(70)000288在文件中的字节顺序反转为88 02 00 70。查找到以后，便可直接将2C 10改为26（pop）00（nop），如图9.14所示。保存后运行，和第一种方法效果相同。

[image]

图9.14　在UltraEdit中查找IL代码

有时，无论采取哪种补丁方式，都会造成程序运行失败，这时很有可能是程序中运用了强名称或其他一些保护方式，当文件被修改后，这

些保护方式会产生异常，从而导致程序无法正常启动。

9.4　代码保护技术及其逆向

本节主要分类介绍笔者完稿时为止出现的各种.Net代码保护技术，内容包括它们的实现原理、保护效果及拆解方法。

9.4.1　强名称

强名称（StrongName）是.Net提供的一种验证机制，主要功能有两个：标识版本和标识原作者。前一个功能用来弥补Windows中DLL机制的缺陷，因为不同版本的DLL，只要强名称不同，便可在.Net中共存；后一个功能主要帮助用户验证自己得到的程序是否为原作者所写，而没有被人修改过（如添加恶意代码）。强名称的原理和Win32下的自校验有点类似，也是利用特定的算法对程序进行hash计算，但检验过程由.Net平台实施。在.Net平台刚诞生时，强名称更多地被人当作一种代码保护机制运用：保证自己的程序不被patch。

先来看一看怎么给程序签署强名称。写一个简单的C#程序，在文本框中键入如下代码后保存为sample1041.cs。由于i的值始终为1，因此输出永远都是“i = 1”。

[image]

在SDK命令行中运行：sn -k mykey.snk，会生成名为mykey.snk的密匙文件，下面用它给sample1041签属强名称。在命令行里输入：csc/keyfile:mykey.snk sample1041.cs，编译成功。运行后仍是输出i = 1。

用ildasm打开exe，双击manifest（清单），会看到.assembly sample1041的如下定义：

[image]

回顾前文描述MSIL时，示例代码中.assembly{}里没有包含任何内容，而这里多出了.publickey和.hash algorithm，这两项便是该Assembly的强名称与所用算法。试着修改一下，将文件偏移02e0h处的17h改为18h，即将源代码中if(i＝＝1)改为if(i＝＝2)，保存后运行，弹出报错窗口，而命令行中会显示出错信息（如图9.15所示）。很明显，“Strong name validation failed.”提示该文件强名称验证失败。

[image]

图9.15　强名称验证失败的报错信息

要对此类文件进行打补丁，必须首先去除强名称的干扰。修改被签署强名称的程序有以下三种方法：移除强名称，重新签署强名称，给系统打补丁。

先说移除的方法，回顾前文的PE结构章节，一个文件中有4处标识指出了该文件是否有强名称：

（1）CLR头中的flags位，去除

COMIMAGE_FLAGS_STRONGNAMESIGNED标志；

（2）CLR头中的StrongNameSignature，RVA与Size均应为0；

（3）Assembly表中的Flags项，减去0001h（PublicKey标识），通常改变后的标识变为0000h（SideBySideCompatible）；

（4）Assembly表中的PublicKey项，指向#Blob的偏移，用0填充。

可以用工具直接移除强名称，如Strong Name Remove，用它打开sample1041.exe，单击Verify后显示如图9.16所示，窗口下部内容是本Assembly中的强名称信息，上部列表框中是所有引用的Assembly的强名称。这里只引用了一个系统文件mscorlib。对于大型程序，如引用框中列出多个Assembly并附有强名称，则应该全部patch，系统文件除外。

[image]

图9.16　用Strong Name Remove移除强名称

对于可以用ildasm反编译的程序，可以在IL代码中将.publickey项和.hash项删除后再重新编译。

关于重新签署（替换）强名称，也有现成的工具，比如国外Libx编写的RE-Sign，某些时候替换比直接去除的通用性更好，建议使用。而给系统打补丁的方法，在看雪论坛里已经由lccracker详细介绍过了，在此一并略过。最后值得注意的是，某些程序在代码中将强名称用于加解密计算（通过GetPublicKey或者GetPublicKeyToken），这时如果仅仅去除它则会出错，应该选择替换强名称并修改相应代码。

由于强名称的最初目的是用于版本识别和代码完整性校验，而非程序保护，所以它的保护强度看起来不值一提。但作为程序开发者，仍应该给自己的所有文件签署强名称，并学习正确的使用方法（如Code Group Strong Name，请参考MSDN）。关于在程序开发中正确使用强名称的内容已超出本书范围，请有兴趣的读者自行深入。

9.4.2　名称混淆

.Net诞生之初，有玩笑说微软在变相搞开源，因为所有的程序信息都被保存在文件中（函数名、字符串、类结构，就差注释了），并可以很方便地用反编译工具还原。为了改变这一状况，混淆器（Obfuscator）应运而生，它可以将所有（类、方法、属性等）名称变为无意义的字符。与Visual Studio捆绑的DotFuscator免费版也许是最早的商业混淆软件，现在已经出现了多种强度更大的Obfuscator，如9rays Spices.net、Xenocode PostBuild、{smartassembly}和DotFuscator正式版等。

最简单的名称混淆原理是改变PE文件中#Strings流的数据，该流中保存了所有的类、方法、属性等的名称，以增加反编译代码的阅读难度。实践出真知，下面来演示一下实现这种混淆的方式。

键入如下代码，保存为sample1042_1.cs后编译。该段代码模仿了最简单的密码验证，判断用户输入的密码是否为“sample”，然后输出验证结果。

[image]

代码中有个函数名叫CheckValid，望文生义，看到这个名称就知道它是比较注册码的。用Reflector打开sample1042_1.exe，程序所有的结构一清二楚。下面修改一下程序，用十六进制编辑工具打开exe文件，来到编移440h处，动一些手脚，将图9.17左边所示的数据全部改为20h。

[image]

图9.17　进行名称混淆（左图：修改前，右图：修改后）

再用Reflector打开exe文件，看一下修改前后对比。如图9.18所示，名称全都为空，特别是那个CheckValid，因为它们全部被替换为了空格的ASCII码（20h）。这就是名称混淆的最基本原理，改变了#Strings流中的字符串。比较成熟的混淆器各有各的修改方式，如将名称全部变为类似x0412vaf84j21lfiavj的无规律数值，或是直接将字符替换为不可打印的ASCII码。读者也可以直接将上面的字符修改为00（即把所有的名称都删除），程序仍能正常运行。为什么呢？因为元数据MethodDef表中已经定义了每种方法的代码偏移和大小，只要这两项数据正确程序便可正常运行，名称是什么无所谓。

[image]

图9.18　名称混淆前后反编译效果对比

不过，有一些函数名称是不能修改的，比如每个类的.ctor和.cctor，系统方法如Console.WriteLine等。一旦这些名称被修改，程序运行将报错。

现在改变一下代码，将CheckValid方法放在一个dll中。新建sample1042_2_lib.cs，键入下面的代码，用如下命令将其编译为一个dll文件：csc/target:library sample1042_2_lib.cs。

[image]

原sample1042_2.cs的代码也做相应改变（粗体为主要变动部分），编译命令为：csc/r:sample1042_2_lib.dll sample1042_2.cs，即命令行中要添加对dll的引用。

[image]

重复前述步骤对exe文件进行混淆，简单起见，只改变CheckValid，将其全部用“20”覆盖。保存后运行，报错了，如图9.19所示。

[image]

图9.19　只修改exe文件的方法名称时，运行的报错信息

错误信息中提示找不到“　　　　”（全部是空格）这个方法。造成这个原因很简单，exe中的方法名改变，dll中相应的方法名也应该改变，并且要和exe的改法一致。用UltraEdit打开dll，将其中的“CheckValid”全部用20h替换，再运行，一切又恢复正常。引用名称必须保持一致（欲知原因，请看MemberRef表的结构），这也解答了为什么混淆过的程序中凡是有关系统方法的调用一律使用原名，因为一般情况下不可能去修改系统方法的名称。因此，系统调用往往是逆向程序的突破口。

保护软件自然不会轻易让逆向者通过系统调用进行分析，比如Spices.net提供一个叫Anonymization的方法，就可以让寻找突破口的难度增加。以9.3节的CrackMe为例，被Spices.net混淆后的代码如图9.20的右图所示，而左图是原代码。

[image]

图9.20　Anonymization方法保护代码（1）

同样是buttonTry_Click事件，混淆前调用了两次MessageBox，最后调用crypt1，混淆后却什么代码都没有了，只剩下一个奇怪的调用“Form1.σ.6”。原来的代码呢？前面说过，形如MessageBox之类的系统方法名是不能被改变的，那它隐藏在哪儿呢？秘密就在这个奇怪的σ.6中。点击6跳转到该方法处，进入后见到了熟悉的用户字符串，但仍然没有见到MessageBox的调用（见图9.21左图）。再次点击字符串前的6，终于找到了目标（见图9.21右图）。Anonymization没有改变系统调用的名称，只是将其隐藏了起来（通过添加新的类和方法）。这种保护手段确实增加了分析的难度，但无论怎么变，系统调用也是无法隐藏的。

对于其他没有介绍的混淆原理，读者要记住一点：必须保证程序的完整和有效性，也必须遵守.Net的规范。

[image]

图9.21　Anonymization方法保护代码（2）

名称混淆的原理解释清楚了，可怎么对付名称混淆呢？通常，名称混淆不会影响静态分析，但如果混淆强度已经达到影响正常分析的地步，则需要考虑修改名称。对于将名称混淆为不可打印字符的，应将这些名称替换回可打印的字符，就是上面修改sample的逆操作。由于替换操作本身是不可逆的，因此并没有办法将所有名称还原为初始状态，只能由读者根据分析结果来猜测名称。

对于包含数个dll文件的大型程序来说，单独修改主程序的名称会造成程序不可运行，原因之一就是dll文件中的名称也必须做相应的改变。看雪论坛dreaman编写的名称反混淆工具可以直接将某个目录下所有的exe与dll文件进行相对应的名称修改，rick编写的名称编辑工具Meta Editor用来修改单个程序集也很方便。另外，现在的反编译软件大多提供了不可打印字符的显示，比如Spices.net可直接输出这些字符的十六进

制名称，Decompiler .Net则是在方法中随机对这些混淆字符串进行重命名，dis#对于单个文件更是支持所有的名称自动反混淆并输出为C#工程文件。有一点需要了解，混淆不能保护原代码，只会增加原代码的阅读难度。

9.4.3　流程混淆

顾名思义，流程混淆就是指打乱程序流程，隐藏原作者的意图，增加代码阅读难度。按混淆的层次可分两种：方法（或类）级别的混淆和IL代码级别的混淆。

关于方法级的流程混淆，前一节“名称混淆”中提到的Anonymization实际就属于该类保护。建立新类作为wrapper，将所有的内部调用和系统调用全部包装到新增的类里。

代码级的混淆就是将原方法的IL代码次序打乱，以达到增加IL阅读难度的效果，同时它还可以防止反编译工具直接将代码反编译为高级语言的形式。听起来很像Win32下的花指令，但由于IL语言本身的特点，.Net中的流程混淆远没有asm下花指令效果好。下面请读者动手实现一个最简单的流程混淆。

用ildasm打开sample1041.exe，导出为dump.il，注意选择导出选项时将Line Numbers选中。其他很多选项是dump元数据的，暂时用不着，如图9.22所示。

[image]

图9.22　ildasm导出IL时的选项

下面修改dump.il，由于sample1041.exe是被加上强名称的，这里正好演示如何在IL中去除强名称。下面只列出修改过的Main方法代码，其中斜体部分是修改过的代码，仅仅是给程序增加了两个br.s跳转。

[image]

在SDK命令行中运行：ilasm /resource = dump.res /output = sample1043.exe dump.il，编译生成sample1043.exe。试运行下，正常，说明强名称已经移除。用Reflector加载，以C#方式查看Main方法的代码。出现什么情况？Reflector报错了，如图9.23所示。

[image]

图9.23　Reflector对流程混淆过的程序报错

不仅是Reflector，现有的反编译软件均不能把代码还原为C#或其他高级语言格式，但IL代码仍是可以反编译的。这种混淆有什么用？由于C#的可读性远远强于IL，这种流程混淆避免了程序直接被还原为高级代码，从而增加了代码阅读的难度。

商业加密软件的流程混淆方法自然比上面复杂得多，如Xenocode Postbuild，它会在源程序中添加许多垃圾代码，将程序分为很多代码

段，运行时不停地进行跳转，并加入大量垃圾代码。最常见的如直接跳转：

[image]

或者是加入恒成立的条件跳转：

[image]

有什么办法反流程混淆？第一种方法，用工具帮助进行流程分析，如IDA提供的流程图功能。用IDA pro加载exe后，按F12键显示Main方法的流程图如图9.24所示。在这个简单的流程混淆中，IDA的表现可圈可点，它将两个br.s跳转连成一线，清楚地显示出了程序的流程。至于对非常复杂的程序实战性强否，留待各位读者在使用中自行体会。

[image]

图9.24　IDA pro的流程图显示

第二种反流程混淆的方法要求有一定的编程功底。一般每种保护软件都有固定的流程混淆算法，如果可以分析出它的算法，便可以写出通用的反流程混淆软件，将反编译出的IL代码进行重组。有时，一些加密软件的混淆有固定的模式（Pattern），看似复杂，却可以通过简单的删除或替换IL指令来反混淆，看雪论坛huweiqi编写的Simple Assembly Explorer便具有基于模式的反混淆功能，读者可自行试用。

第三种方法，无论名称混淆还是流程混淆，代码还是摆在面前了，不过这次又多穿了一层马甲。因此，耐心的分析是对付这类保护方式的杀手锏。

9.4.4　压缩

随着.Net可执行文件的流行，出现了一些支持.Net的加壳软件。本节讨论的是这些加壳软件中不含元数据加密功能的那一类（也可称为压缩壳），至于含加密功能的壳的分析，放在下一节。

由于笔者所了解的几个壳对.Net 2.0支持不是太好，所以本章程序运行在1.1平台下，无论在哪个版本的.Net下，其基本原理是相同的。先看示例程序代码：

[image]

[image]

代码的功能就是通过Reflection空间Assembly类的GetEntryAssembly方法，取得正在运行的Assembly全名并输出（一个程序集的全名包括名称、版本、语言和PublicKey标识）。

.Net程序的压缩壳按编写方式，可分为基于.Net平台编写和基于Windows平台编写两大类。纯.Net编写的压缩壳包括Sixxpack（现更名为AdeptCompressor）、.NETZ和bsp。Sixxpack将原程序压缩后存在新

文件里，运行的时候动态解压至byte[]数组中，最关键的是这两句。

[image]

.Net从诞生开始就支持这种内存中的Assembly载入，然后Invoke调用该Assembly的方法。解压的方法也有几种：手动dump、用工具dump、根据解密算法直接还原原程序等。

bsp也利用了Sixxpack类似的原理，关键是压缩算法不同。这里就用bsp压缩sample，然后讲一下手工dump。用PEBrowseDbg载入压缩过的程序，在三个方法上全部下断，然后继续运行，如图9.25所示。

[image]

图9.25　bsp压缩的程序

第一次中断在token为06000001的方法处，在代码窗口中浏览一下，可以看出bsp用的也是Assembly.Load，然后对入口method进行Invoke，不过它对原程序进行了混淆，因此调用的入口方法不是sample中的Main，而是一个名称为不可显示字符串的方法。此时的任务：等原程序全部解压完毕后进行dump。向下看，来到IL_02D1处的第一句asm代码，然后“Run to selection”。

[image]

注意斜体的那句指令（在不同的机器上地址可能不一样），对照IL代码，这句应该是将byte[]数组地址传递给ecx，作为Assembly.Load的参数。双击esi，查看该处的内容，如图9.26所示。熟悉的字符出现了，这不就是PE头吗。

[image]

图9.26　esi所指内存的数据

可是这并不是原程序，而是bsp的一个loader，通过这个loader再调用原程序。是否dump该loader与本程序无关，但dump方法还是值得介绍下，基本流程如下：由于PEBrowseDbg没有dump的功能，因此利用WinHex将数据保存为文件。用WinHex打开进程的内存，按“Alt + G”键跳转到01296FA0h（01296F98h + 8，注意这个值不是固定的，而是随机分配的内存地址）处，然后定义块首Begin of block，问题是块尾定义在哪。注意刚才esi所指的内存，并不是直接指向了“MZ”，前面还有两个数据。与Win32下不同，.Net中任何托管代码都不是直接与内存地址打交道，byte[]数组也是一个系统类型，而它的第2个DWORD就指明了它的大小为5000h。这个猜想对不对，不如保存后再验证。因此，块尾定义在1296FA0h + 5000h = 129BFA0h。保存为文件后，用Reflector载入，原来只是个loader而不是原始exe文件，且它是一个dll，不能直接运行。

下一步该怎么办？下断点。由于被加壳的程序是exe，里面含有

entrypoint，可以猜想loader是通过调用这个入口方法执行原exe。而要调用该方法，首先要取得该入口方法的“地址”。因此，将断点下在mscorlib的System.Reflection.Assembly.get_EntryPoint。点击运行后，果然中断下来，如图9.27所示。

[image]

图9.27　中断在get_EntryPoint

按F10键步过ret，返回后来到如图9.28所示代码处（PEBrowseDbg中调试快捷键与OllyDbg不同，走的是微软路线），中断在MOV EBX,EAX一句，它的上一条CALL指令便是调用下断点的方法（get_EntryPoint）。

[image]

图9.28　调用Assembly.GetEntryPoint返回后的代码

如果继续跟下去，还可以跟到sample1044中的Assembly.ToString方法，不过这里已经可以dump了，因为取得入口点的调用只能出现在完整的Assembly解压之后。用WinHex重新打开内存，查找“BSJB”（CLR头的标志，要熟练运用哦），当来到1221270h处时，发现离该标志不远的字符串中出现了“sample1044”，这不就是#Strings流吗。

[image]

再从“BSJB”向上寻找PE头，越过“.text”节，来到01221000处，这里便是需要dump的内存块的头部了。块的大小又如何选择？由于内存区域一般成片分配，在PEBrowseDbg中选择Index Detail看一下内存块详情（如图9.29所示），01220000h到01222FFFh是一个块，就先dump这一段。同样的方法，用WinHex保存为文件，改扩展名为exe，这就是原始程序。

[image]

图9.29　PEBrowseDbg中显示的内存块

上面讨论了手工还原bsp压缩的Assembly的方法，希望大家举一反三。能够直接dump整个Assembly，这也是为什么一直说bsp是压缩壳而不是保护壳的原因。

也有将压缩壳保护做得比较好的商业保护软件，如{smartassembly}，该壳将多个文件加密后保存，并且使用了强名称作为解密密钥，且结合强大的混淆，因此强度不错。

传统的壳也有支持.Net PE文件的，它们加壳的最大特点是，生成的文件是Win32而不是.Net。压缩壳不是.Net保护技术的主流，本书就介绍到这。像这类将原文件经过运算存储在文件中（或资源中），运行时

将完整原文件在内存中展开并运行的加壳方式叫Whole Assembly Protection。这种“整体程序集保护”的方式被证明强度是非常弱的，它可以轻易地被NET Unpacker之类的脱壳软件dump。

9.4.5　加密

如果一个程序声明必须在.Net下运行，但用PEiD检测却是Win32程序，多半这个程序已经被加密了。.Net保护技术发展到现在，最流行的保护措施之一便是利用Win32的本地代码通过加密元数据、挂钩系统内核等手段保护原程序。要分析被这类程序保护的.Net文件，读者不仅要对.Net平台非常熟悉，还必须对.Net内核有一定了解，比如JIT的编译过程，执行引擎EE（Execute Engine）的原理等。所以在介绍加密前，有必要先简要叙述一下本章用到的内核知识。由于层次的“降低”，本章所用的调试器也从IL级的PEBrowseDbg转为汇编级的OllyDbg。

如果用Win32的PE工具打开.Net PE，会发现整个引入表只有一个API，exe对应mscoree.dll的_CorExeMain；dll对应mscoree.dll的_CorDllMain。这就是说，Windows的loader载入.Net PE后，只负责跳转到相应的dll中，随后该程序便运行在ee的监管中，Windows本身不再负责该程序的内存分配、线程管理等工作，而将这些工作交给了.Net框架。

用OllyDbg任意跟踪一个.Net PE的加载过程，将模块加载的跟踪选中，观察dll的加载顺序。mscoree.dll一直存在，随后第一个.Net组件是mscorwks.dll，然后是mscorlib，再接着是mscorjit（其间会加载一些本地dll）。如果是窗口应用程序还会出现一些和Forms相关的dll，VB编写的程序会有Visual Basic的相关dll。.Net的核心代码在mscorwks.dll中，而JIT部分主要由mscorjit.dll负责。当遇到没有编译的方法时，mscorwks调用JIT把代码编译为asm后，将控制权交给asm并执行，完毕后mscorwks再收回控制权。正因为mscorwks和JIT在整个.Net中的核心地位，大多数加密软件都以这两个dll作为突破口，或进行挂钩，或进行包装，这种种操作的最终目的就是在JIT前将被加密的IL代码和元数据恢复为正常，并在方法结束后再将元数据和IL代码销毁，以达到保护原程序的目的。

本节分析的第一种保护方式是CodeVeil，先查下壳，PEiD显示是UPolyX v0.5，是否准确无关紧要，用OllyDbg加载后查看可执行模块，没有mscoree.dll。.Net程序怎么会没有这个必需的dll呢？遇到这种情况，熟悉Win32下脱壳的读者应该很自然地想到一个经典断点：LoadLibraryA(W)。按F9键运行后，第一个中断便是LoadLibraryA，参数FileName为“mscoree.dll”（如图9.30所示）。

[image]

图9.30　在加载mscoree.dll时中断

跟踪至LoadLibrary返回并执行用户代码，不一会就看到利用GetProcAddress取得_CorExeMain的地址，看来CodeVeil把原先.Net程序自动完成的工作给“手动化”了。

再向下会调用VirtualProtect，壳可能要开始解密了。

[image]

看一下堆栈，数据为402000h，这是.text区块的内存地址。

[image]

前面介绍PE结构时说过，.text区块保存了所有的元数据和IL代码，.text区块的内存地址为402000h，先看一下该处的内容。

[image]

和未加密的CrackMe比较，你会发现这明显是加过密的数据。由于壳最终肯定会在某处将402050h处的数据解密，此时便可以在该数据上设置断点，从而避免在无尽的花指令中跳来跳去。下内存断点会改变原数据，使还原的数据出错，因此在第一个字节处下硬件访问和写入断点。取消所有已下内存断点后运行，不一会就停在硬件断点上。

[image]

这段代码完成的就是第一次解密。第一次解密，难道还有二次解密？此时，402050h处的数据如下，通过与原程序比较得知现在是完全解密的数据：

[image]

保持断点不变，继续执行，不一会就来到第二次硬件中断处：

[image]

不多见的SSE指令。这段代码执行完毕后，看一下IL代码处的数据。

[image]

前后两次数据有相同的也有不同的，相同的是方法头，不同的是方法体。至此，CodeVeil保护的秘密真相大白：每次执行方法前，CodeVeil将方法体的IL代码还原；每次方法执行完毕，再将IL动态加密，防止dump，但方法头保持不变。

什么是方法头，什么是方法体？原来，.Net的方法在内存中是按一定结构存在的，主要由三个部分组成：方法头、方法体、异常处理表。

.Net中的方法有两种，fat和tiny，两种方法结构不同，定义均在CorHdr.h中。.Net规定，只有当方法中没有异常处理表，堆栈深度小于8，并且代码大小为64字节之内时，方法才可为tiny。

[image]

看一下未加密程序文件偏移402050h处第一个method的头结构：

[image]

这是一个fat header，表9-11说明了这些数据的具体含义。

表9-11　Fat Method Header结构

[image]

读者应该形成一个条件反射，看到内存数据为13 30h就要想到有可能是方法头，并且是fat的。在实际情况中，遇到fat方法的几率也远大于tiny。

回到OllyDbg中来，究竟什么时候可以dump呢？既然知道了CodeVeil的运行原理，时机就很好掌握，在CodeVeil将元数据全部解密完毕后，在加密元数据之前。错过这个时机dump下来的可全是乱码。

下面准备dump了，前面介绍过了手工操作，这里使用工具Task Explorer（集成在Explorer Suite中），操作见图9.31。在进程中选中CrackMe，在下方Module框中选中CrackMe模块，在右键弹出菜单中选择“Dump PE”。

[image]

图9.31　在加载mscoree.dll时中断

用Reflector载入dump下来的文件，显示没有CLR头，不是.Net PE文件。剩下的工作是文件结构修复，这里介绍两种半自动的修复方法。第一种方法，用CFF载入后修复PE，重新对齐文件，保存后已经可以用ildasm载入了，如图9.32所示。ildasm反编译，ilasm再编译，一个完好的.Net PE文件生成了。

[image]

图9.32　用CFF修复PE文件

第二种方法，用dis#载入脱壳后的文件，首先进行反混淆，如图9.33所示。有时，太多的不可打印字符会使编译工具出错。

[image]

图9.33　dis#反混淆前后对比

对比dis#反混淆前后的效果，原先的乱码已经被替换为Class之类的名称，虽然只是简单的改变，但可读性已大大增加（见图9.34）。

[image]

图9.34　dis#反混淆前后对比

随后，在dis#中将文件导出为C#工程文件（如图9.35所示），便可以用Visual Studio载入了。将入口方法改为Main，再次编译，于是exe文件便生成了，运行检测一切正常。（注：现在大多数程序均经过流程混淆，可直接导出C#并成功编译的机会不是很多。）

[image]

图9.35　dis#导出C#工程文件

可以看出CodeVeil是一款比较“懒”的程序，因为不论是哪个方法调用JIT，它都会将所有的元数据进行解密，现在已经有壳做到了per-method解密，就是每调用一个方法时解密该方法的元数据，这样就无法轻易地完整dump整个程序集。

下面来看看国外另一款壳的保护方式。被该壳保护的产品可以用Reflector载入，但是所有的方法都为空（见图9.36）。很明显，这类方法的IL代码在运行时要被动态释放。

[image]

图9.36　被保护的程序可以用反编译程序载入，但方法都为空

注意一下程序的安装目录，会发现一个rscoree.dll，如果说CodeVeil的秘密稳藏在exe文件本身（.rsrc节），那么该壳的秘密就稳藏在这个本地dll中。

Reflector打开后，发现只有一种方法体的IL代码不为空，这就是静态构造函数.cctor，如图9.37所示。同样是frmLogin类，.cctor的方法却包含一个到 < PirvateImplementationDetails > 的调用。静态构造方法在类被加载时执行，且只执行一次，它的执行顺序先于类中所有的代码。但就是这一次执行，已经足够壳解密该类的所有代码了（见图9.37）。

[image]

图9.37　静态构造函数里包含调用解密的代码

点击cctor中的链接，来到 < PrivateImplementationDetails > 的内部，最有意思的代码出现在_RSEEStartup方法中：

[image]

这里的代码属于.Net中的互操作，是引用本地dll中函数的声明，这个本地dll正是刚才在程序安装目录中看见的rscoree.dll。由于调用了该dll的导出函数“_RSEEStartup”，下面的分析便是进入dll里看这个启动函数到底做了什么。

用Win32反汇编软件对rscoree.dll反汇编，进入startup函数后，不一会见到如下代码：

[image]

跟踪到GetModuleHandleA时，可以从参数里看出“vvn8~stlym}w”的解码字符串为“mscorjit.dll”，下面看壳对JIT做了哪些手脚。

[image]

这次解码出来的名称是getJit，getJit是mscorjit.dll为数不多的导出函数之一。取得该函数地址后，紧接着调用它，然后保存getJit的返回值。有趣的是保存getJit的返回值完毕后的一段代码：

[image]

InterlockedExchange将eax指向的第一个双字值变为了10024850h处的值，也就是100027F0h，其中eax为getJit的返回值。现在的问题就是getJit返回了什么？

用IDA反汇编.Net内核文件mscorjit.dll，来到getJit函数的代码处：

[image]

粗体的mov指令说明了eax的返回值是CILJit::vftable的偏移，很自然，想到查看vftable的内容是什么。双击后来到vftable的偏移处（就在getJit下方）：

[image]

vftable偏移处是一串指针列表，看名称应该是一些系统函数的地址。到这里，rscoree的流程就很清楚了：取得一系列系统函数的地址，并对其中的某个地址进行替换。由于它只通过InterlockedExchange替换了第一个指针，因此只需要弄清vftable的第一项是什么：

[image]

这个函数是JIT引擎的核心函数compileMethod，它的输入是IL代码，输出为本地代码。只要是.Net程序，就必调用该函数，因此该地址可以称为.Net下的万能断点。rscoree将ICorJitInfo中指向compileMethod的指针替换为dll中的100027F0h处，暂且把它称为hookcompileMethod，这样在每个JTT事件时，.Net都会调用hookcompileMethod，从而使壳有机会解密代码，再将解密后的数据传递给真正的compileMethod。运行OllyDbg，下断在hookcompileMethod，跟一会就会看到如下代码，其中最后的call就是调用最初在_RSEEStartup中保存的原始JIT方法地址。

[image]

到这里，前一种壳CodeVeil的秘密也清楚了。CodeVeil是如何挂钩JIT的呢？在OllyDbg中查看CodeVeil保护程序的ICorJitInfo接口处的数据，结果数据显示其中第一个双字指向了主程序内部，compileMethod被挂钩了。

[image]

到这里，读者已经初步接触了壳挂钩.Net内核的操作原理。其实CodeVeil和第二种壳的挂钩方法最多只能称作Wrapper，更强的加密方法是Hook内核dll，改变其代码流程，从而在.Net内核JIT的过程中进行加密与解密。这种加密方式的强度远大于Wrapper，有兴趣的读者可以自行深入分析。

对付此类保护的方法，可以利用进程注入后反射的方法取得源代码和元数据，再重构PE文件，或者采用更通用的方法，即挂钩JIT层得到代码。可以看出，单纯使用加密壳的安全程度仍然一般，程序开发者应尽量将混淆和加密结合使用。

9.4.6　其他保护手段

上面介绍的是5种主流保护方式，除此之外，程序中还经常应用一些小措施来增加逆向难度，比如反调试跟踪、网络验证等。下面就介绍其中的一部分小技巧。

1．反监测

程序保存注册信息时，最忌讳Spy软件的监测，因此在进行关键操作（文件和注册表）时，往往会检测是否有Spy软件在运行。最简单的实现是在关键代码段中枚举所有窗口，取得窗口的名称，并比较是否包含敏感字符串。最常见的是Win32API中的FindWindowExW函数。

2．反调试跟踪

Win32下的反调试做得比较完善，可惜.Net是高层平台，无法直接利用寄存器实现Anti，因此纯.Net实现的反调试相对来说功能要弱些。程序被调试时，Win32下有IsDebuggerPresent用来检测，.Net下则是通过System.Diagnostics.Debugger实现，代码如下：

[image]

OllyDbg中有隐藏调试器标志的插件，PEBrowseDbg中也有相应选项，如图9.38所示。因此，这种检测方法效果一般。

[image]

图9.38　PEBrowseDbg中的隐藏调试器选项

另一种反调试的方法也借鉴自Win32，即在两段代码处分别取当前时间并相减，如果差值过大则认为程序被单步跟踪调试了，后面的流程则会调用完全不同的算法来迷惑分析者，或直接让程序退出。下面的示例代码是比较两次时间之差是否大于3秒：

[image]

第三种方法是利用C＋＋/CLI的混合编译特性，在本地代码中加入强大的反调试代码。这种方法较新颖，但本地代码也是可以被patch的。

3．网络验证

网络验证现在已经比较常用，在.Net中主要依靠System.Net中的几个类实现。如果程序没有提供网络功能，而在运行时防火墙不断提示程序要求联网，读者就应该注意分析是否存在网络验证了。

4．虚拟机

VM保护是当今很热门的一种保护方式，Win32下已经出现多个利用VM的保护软件，而.Net中也开始出现类似保护方式。虽然现在尚未普及，但绝对是将来.Net保护方式发展的一个大方向。

5．加密锁

这个大家很熟悉了，主要用于行业软件中。虽然保护的目标是.Net程序，但其核心调用仍属于Win32范畴，与.Net关系不大。

9.5　深入.Net

本节主要介绍.Net中相对较深入的内容，包括反射机制、代码文档对象模型的应用、.Net的几个核心接口，以及通过开放源代码的简化.Net框架及框架内核dll来研究.Net核心的运作方式。本节的内容看似和加密、解密没有直接关系，但掌握本章的内容会让读者的逆向分析技术更上一层楼。

9.5.1　反射与CodeDOM

什么是反射（Reflection）？反射能做什么？

MSDN中的解释很清楚，这是.Net中获取运行时类型信息的方式，.Net的应用程序由程序集（Assembly）、模块（Module）、类（Class）组成。而反射提供一种编程的方式，让程序员可以在程序运行期获得这几个组成部分的相关信息。例如：Assembly类可以获得正在运行的程序集信息，也可以动态地加载程序集，以及在程序集中查找类型信息，并创建该类型的实例。Type类可以获得对象的类型信息，此信息包含对象的所有要素：方法、构造器、属性等，通过Type类可以得到这些要素的信息，并且调用之。MethodInfo包含方法的信息，通过这个类可以得到方法的名称、参数、返回值等，并且可以调用之。诸如此类，还有FieldInfo、EventInfo等，这些类都包含在System.Reflection命名空间下。

从逆向的角度看，通过反射，程序可以做下面的事：取得当前“进程”中加载的所有Assembly，取得Assembly的各个Module和Type（例如Reflector的树状结构），取得某个方法的IL代码，动态载入新的Assembly，执行某个指定方法。

下面来看几个应用反射的例子。第一个例子是rick写的利用反射取得本程序所有IL编码的示例。程序没有加密，可直接在Reflector中参考源码。

首先运行未加密的版本，得到的输出片段如下：

[image]

下面是CodeVeil加密过的程序，运行后dump到的IL代码：

[image]

除了方法的名称因为混淆而改变外，IL代码的字节并没有改变。不过CodeVeil加密过的程序，只会在第一次反射时取得正确的元数据，第二次则会出错。原因就是CodeVeil会将运行过的IL再次加密，第二次执行反射时，由于JIT后的代码已经存在，则不再次调用解密，因此只能得到被加密的元数据。

现在的关键问题是，怎么样让一个程序执行解密需要的反射代码呢？这需要用到另一个程序injectDll，这个程序功能就是通过注入dll，在另一个程序空间里执行用户编写的C#代码。这个程序作为脚本演示平台非常方便，下面演示一段脚本，功能是取得String的所有方法的内存地址。

[image]

脚本输出片段如下：

[image]

这个脚本有什么用呢？试想在OllyDbg中对取得注册码长度的代码下断点，便可直接在7A06C32Dh处下断点跟踪，很方便。

利用注入原理和.Net的反射机制，可编写出很具有实战性的工具，如injectReflector，可以实现无须dump直接在内存中查看被加壳程序的IL代码。如图9.39所示，左图演示用Reflector直接载入被加壳的程序出错，右图演示利用injectReflector通过注入反射便可直接显示IL。对于Win32程序，同样可以注入，之后便可以在普通Windows程序中调用.Net的类和方法，是不是很有趣！更多反射的功能，由大家自己摸索，网上相关资料很多。

[image]

图9.39　利用注入反射可以取得被加壳程序的源代码

那么injectDll是怎么做到内存中动态执行C#脚本的呢？.Net为程序员提供了代码文档对象模型（CodeDOM）来实现该功能。关于CodeDOM的资料也是公开的，有兴趣的读者可以自行阅读。

有时，injectDll注入反射脚本并不能取得所有的程序信息，特别是被加密过的程序。这是因为反射的底层是调用了非托管元数据API接口，此类接口直接在内存中取元数据，而不管其是加密的还是解密的。

9.5.2　Unmaganed API

在1.1版.Net的安装目录下，有一个Tool Develop Guide目录，里面的doc子目录包含了面向.Net工具开发者的相关文档，samples目录则对应于文档的一些代码示例。微软对这些文档的描述是“开发人员工具指南

包含生成在.NET Framework中运行的底层开发工具（如编译器、浏览器、分析器和调试器）的文档、产品规范和示例”，由此可见这些文档和代码的含金量。2.0版中，这些文档已经被移至MSDN。

Unmanaged API是.Net平台提供的一系列非托管API接口，核心是运用了COM技术。这一系列的API很多，其中最重要的是三类：调试接口（ICorDebug类）、分析器接口（ICorProfiler类）和元数据接口（IMetaData类），而实际使用过程中，这三类接口往往会相互调用。正因为它们的重要性，1.1版的文档中只包含了对这三类接口的详细描述而省略了其他内容。

先从分析（Profiling）API说起，通过ICorProfiler接口提供的方法，可以得到以下过程的相关信息并控制它们：Application的开始/结束，Assembly的载入/卸载，Methods的开始/结束，Module的载入/卸载，Class的载入/卸载，与COM接口的互操作，托管/非托管代码的即时编译等，基本上.Net的所有核心操作都可以通过Profiling API来得到信息。中文MSDN中提供了非常好的一篇示例文章：《在.NET Framework 2.0中，没有任何代码能够逃避Profiling API的分析》，附带的源码很有参考价值。看雪论坛已有数篇文章讲述怎样在现有代码的基础上实现自己的Profiler，并利用该Profiler动态地修改内存中的IL代码，这里不再赘述，只介绍一些基本概念。

Profiling API是通过编译成dll文件，注册为系统的COM服务而实现的。因此，Profiler的例子通常都是三个文件：ProfilerCallback.h、ProfilerCallback.cpp和Profiler.cpp。前两个是实现Profiler的核心功能代码，最后一个只是实现了COM dll的基本框架。通常要实现自己的Profiler，只需要在ProfilerCallback.cpp中添加相应代码即可。

当一个Profiler的dll编译成功后，要想正确使用必须有下列步骤：

① regsvr32 Profiler.dll（注册服务）。

② 设置进程的环境变量，主要是两个：

[image]

③ 运行需要监测的程序，得到Profiler的输出。可以输出到调试器，也可保存为文件。

④ 关闭Profiler，set Cor_Enable_Profiling = 0x0。

⑤ regsvr32/u Profiler.dll（卸载服务）。

一般说来，Profiler主要提供监测的功能，程序开发中通常用它来检测程序执行效率。若想动态修改IL代码，则应注意方法头和方法体的结构：对于不含异常处理表的fat方法与普通的tiny方法，直接修改即可，而含有异常处理表的方法则要注意调整异常处理表的数据。

在一个Profiler中，不可能不用到元数据接口。比如在代码监测过程中，需要修改程序让它动态加载一个dll并执行其中的方法，这就需要引用该dll（Assembly）的某个类型（Type）的某个方法（Method），而程序集、类型和方法在.Net中的引用都需要通过元数据定义来实现。看一段代码示例：

[image]

该段代码用在Profiler中，首先通过ICorProfilerInfo接口取得IMetaDataEmit接口，继而通过后者获取IMetaDataAssemblyEmit接口，为后续代码动态定义Assembly引用做准备：

[image]

DefineAssemblyRef后，运行中的程序便被动态添加了“inject”的程序集引用，也就是新添加的程序集拥有token了。同样的方法，继续为类和方法构造token值并定义引用，便可在代码中调用该方法了：callmethodToken。

光盘映像文件中提供了一个只监测方法入口参数和返回值的Profiler，按照上述步骤执行后，监测结果保存在output.log文件中。MSDN提供了更多开源的Profiler完整代码，值得参考。

下面的代码片段出自某Profiler软件的监测结果，其中注册码已经被捕捉到（粗体部分）。

[image]

有时带Profiler的程序运行会不稳定，而且所有的加密软件都提供了Anti功能。比如挂钩系统dll，在Profiler获取数据后再解密，从而使监测数据全部是乱码；或者在进程开始时将Profiler必需的环境变量改为无效。因此Profiler并非万能，还需读者灵活使用。

第三个接口是调试器接口。调试器接口通常用于编写调试器，了解它的原理能让调试者更好地在.Net下使用调试器。

ICorDebug接口提供了控制.Net调试器和被调试程序的所有方法，其中两个最重要的方法是ICorDebug::SetManagedHandler()和ICorDebug::SetUnmanagedHandler()，前者用于托管程序，后者用于非托管程序。当被调试程序中发生了感兴趣的事件时，系统便会调用这两个函数注册的处理函数，调试器便有机会进行需要的工作。通常.Net中用到的都是SetManagedHandler，它要求提供一个ICorDbgManagedCallback*的回调接口。其他一些接口如ICorDebugProcess、ICorDebugAppdomain等，均是在调试器中用于接收和控制特定对象的信息。下面的代码片段中，注册的回调函数通过CreateAppDomain事件（当一个AppDomain被创建时调用该方法）便可以得到关于进程和AppDomain的信息。

[image]

从传入参数中取得ICorDebugAppDomain接口并保存在pAppDomain变量中，之后便可以调用该接口提供的方法。下面的代码中，首先是将调试器Attach到该程序域中，然后查看该域中所有的程序集：

[image]

Unmanaged API的内容实在太多，读者学习过程中应选关键点入

手，其中最实用的莫过于元数据接口，因为它提供了直接观测程序元数据的方法，而不用程序编写者纠缠于烦琐的PE结构。举例来说，假设要得到所有的用户字符串，如果没有元数据接口，程序编写者首先要载入PE文件，分析文件头，定位到CLR头，然后在CLR头中寻找#US流的位置和大小，并手动一项项读入字符串。而在元数据接口中，调用一个EnumUserStrings便可以得到所有字符串，是不是很方便！

尽管Profiler和Debugger很方便，但现在的很多加密软件都有如下声明：Anti-profiler、Anti-debugger，它们实现Anti的原理通常是对.Net内核做些小手脚。要想对付这些Anti，需要对.Net内核有一定了解。

9.5.3 Rotor、MONO与.Net内核

要深入.Net的内核，通常有三种方法：查阅微软的文档，反编译.Net内核文件，阅读源代码。第一种方法效果一般，MSDN一般只讲what和how，至于why提及的不多（不过不代表MSDN不重要，它仍是Windows平台开发最好的参考资料）。第二种方法对分析者的逆向功底要求很高，但得到的结论是最权威的。第三种方法最便于掌握内核运行流程，可问题是，去哪里找.Net框架的源代码呢？

由于微软将.Net申请了ECMA标准，因此公开了一些源代码，特别是为了.Net的普及，微软公布了一份简化版的.Net源码Rotor，又叫Shared Source CLI（SSCLI）。虽然不是完整的.Net源码，但依然提供了极有价值的资料。另一个开源.Net平台则是MONO，主要目的是在Linux类的操作系统上实现.Net平台。本小节主要介绍如何利用SSCLI与本机.Net文件的逆向相结合，来探究.Net的内核世界。

Windows平台上的.Net框架核心文件以mscor开头的dll文件是研究重点，其位于“\WINDOWS\Microsoft.NET\Framework\版本号\”中。用IDA pro对感兴趣的文件进行反汇编，IDA会提示是否从网上下载dll的符号文件，这时一定要点yes，带符号反汇编出来的汇编代码可读性大大增强。

通过将Rotor的源代码与相应的反汇编代码进行对比，会看到商业化.Net框架（就是.Net Framework）与开源.Net框架的一些区别。两者的区别不必关心，利用SSCLI来辅助反汇编代码的阅读才是目的。来看几个例子，下面的代码源自IDA对mscorjit.dll的反汇编代码：

[image]

compileMethod方法中的后三个参数都没有提示名称，除了动态跟踪外，很难得知它们代表了什么。再来SSCLI中看相应的代码：

[image]

也是5个参数，其中flags为输入参数，而entryAddress和nativeSizeOfCode是输出参数，SSCLI瞬间解开了刚才的疑问。但到这里仍不能确定SSCLI给出的参数与反汇编中的参数是否一一对应，这需要在动态调试过程中加以确定。（实际证明，SSCLI的答案是正确的。）

除了给出一些未公开的变量与结构外，SSCLI还让调试者更轻松地分析.Net内核的一些运行机制。最基本的，一个exe是怎样被加载的，读者可以在SSCLI里通过跟踪它的代码流程弄清来龙去脉。

再来看SSCLI的另一个应用示例。上一小节提到了某些加密软件可以轻松躲过Profiler的监测，那就在SSCLI里寻找Profiler的实现代码，很自然地来到“clr/src/profile”中。通过阅读代码知道，.Net中EE引擎提供了Profiler接口，这可以通过GetEEToProfInterface得到。再回到Windows中的.Net框架上，其中mscordbc.dll提供了两个导出函数，如图9.40所示。

[image]

图9.40　mscordbc.dll的导出表

这说明了mscordbc.dll负责了EE与Profiler通信的部分工作。下面就是利用OllyDbg调试带Profiler启动的程序，目的是找出系统在何处判断是否需要向Profiler输出信息。最终的秘密存在mscorwks.dll中：

[image]

其中的test语句判断是否需要输出JIT信息到Profiler，如果是则jnz跳转，跳转后会来到如下代码处：

[image]

其中call语句已经注释出了，是调用EEToProfInterfaceImpl::JITCompilationStarted方法，进入该方法中来到如下代码处：

[image]

粗体的那句call指令便是调用Profiler中的代码。到这里，一个Profiler是怎么输出关于JIT开始信息的流程就很清楚了。自然，想让Profiler失效也只是一个简单的内存patch而已：将test后的jnz指令nop掉即可。加密软件能让Profiler失效，调试者让Profiler恢复功能也不会太难。

本节主要介绍一些.Net下的深入内容，从用户态的反射讲到COM接口非托管API，最后是汇编及源代码级别的.Net内核。和本章前面的内容一样，本节主要给读者提出一些分析方法，授之以鱼不如授之以渔，.Net真正的秘密还要靠读者自己去探索。

注释

① 本节由丁益青编写。
② 本章由单海波编写。

第5篇　系统篇

- **第10章　PE**文件格式
- **第11章**　结构化异常处理

PE是Windows上可执行文件的格式，熟知PE文件将有助于对操作系统的深刻理解。如果你知道EXE和DLL里面的奥秘，那么你将成为一名知识更加渊博的程序员。本书用大量篇幅，图文并貌地详细讲解了PE格式。

SEH的出现已非一日，但有关SEH的知识资料却不是很多。SEH不仅可以简化程序的错误处理，使程序更加健壮，还被广泛应用于反跟踪和加密中。本书从调试角度讲述了SEH的机理，掌握这些后，调试SEH处理的程序，就会更加自如。

第10章　PE文件格式

从某种意义上讲，可执行文件的格式是操作系统本身执行机制的反映。虽然研究可执行文件格式并不是程序员的首要任务，但这种工作能够积累大量的知识，有助于对操作系统的深刻理解。掌握可执行文件的数据结构及其一些运行机理，也是研究软件安全的必修课。

在Win16平台上（如Windows 3.x），可执行文件是NE格式。在Win32平台上（包括Windows 9x/NT/2000/XP/2003/Vista/CE），可执行文件是PE格式。PE是Portable Executable File Format（可移植的执行体）简写，它是目前Windows平台上的主流可执行文件格式。

PE文件衍生于早期建立在VAX/VMS上的COFF文件格式（Common Object File Format）。由于许多Windows NT的创始者来自于数字设备公司（DEC），他们很自然地使用已有的代码来快速开发新的Windows NT平台。采用术语“Portable Executable”是因为微软希望有一个通用于所有Windows平台和所有CPU（x86，MIPS，Alpha等）上的文件格式。当然，在CPU指令的二进制译码等方面会存在差异，但最重要的是系统的装载器和编程工具无需对任何一种新出现的CPU进行重写。从大的方面讲，这个目标已经实现。这使得Windows NT和它的后代，Windows 95和它的后代以及Windows CE拥有了相同的格式。

为了将精力集中在Windows NT上，Microsoft公司放弃了当时的32位工具及文件格式。例如，在Windows NT出现之前，16位Windows的虚拟设备驱动程序使用的是32位的文件格式——LE格式。更重要的是OBJ格式的改变：在Windows NT的C编译器之前，所有Microsoft编译器使用的都是Intel的OMF（Object Module Format）规范。在前面讲过，基于32位Windows系统的Microsoft编译器所生成的是COFF格式的目标文件。Microsoft的一些竞争者（如Borland及Symantec）继续使用Intel的OMF格式，放弃了COFF格式。其结果是OBJ和LIB必须针对编译器发送不同的版本。

描述PE格式及COFF文件的主要地方是在winnt.h，其中有一节叫“Image Format”。该节给出了DOSMZ格式和Windows 3.1的NE格式文件头，之后就是PE文件的内容。在这个头文件中，几乎能找到关于PE文件的每一个数据结构定义、枚举类型、常量定义。可以肯定，在别的地方也能找到相关文档，例如在MSDN里，但是winnt.h是PE文件定义的最终决定者。

EXE和DLL文件之间的区别完全是语义上的，他们使用完全相同的PE格式。唯一的区别就是用一个字段标识出这个文件是EXE还是DLL。还有许多DLL的扩展，如OCX控件和控制面板程序（.CPL文件）等都是DLL，它们有一样的实体。

另外，64位的Windows只是对PE格式做了一些简单的修饰，新格式叫PE32＋。没有新的结构加进去，其余的改变只是简单地将以前的32位字段扩展成64位。对于C＋＋代码，Windows文件头的配置使其拥有不明显的区别。

PE文件中的数据结构一般都有32位和64位之分，如

IMAGE_NT_HEADERS32、IMAGE_NT_HEADER64等。除了在64位版本中的一些扩展域以外，这些结构几乎总是一样的。在winnt.h都有#defines，它可以选择适当的32位或64位结构并给它们起成与大小无关的别名（在前面的例子中，可以写成IMAGE_NT_HEADERS）。结构选择依赖于用户正在编译的模式（尤其_WIN64是否被定义）。若没有特别说明，本书的实例都是基于32位PE格式进行研究的。

在跳到PE格式细节之前，仔细看一看图10.1，该图显示了PE格式的大致布局。下面将分别解释每一块的内容。学习的同时，建议用Stud_PE工具配合，该款工具直观地显示出PE各部分数据。

[image]

图10.1　PE文件的框架结构

10.1　PE的基本概念

PE文件使用的是一个平面地址空间，所有代码和数据都被合并在一起，组成一个很大的结构。文件的内容被分割为不同的区块（Section，又称区段、节等），区块中包含代码或数据，各个区块按页边界来对齐，区块没有大小限制，是一个连续结构。每个块都有它自己在内存中的一套属性，比如：这个块是否包含代码、是否只读或可读/写等。

认识PE文件不是作为单一内存映射文件被装入内存是很重要的。Windows加载器（又称PE装载器）遍历PE文件并决定文件的哪一部分被映射，这种映射方式是将文件较高的偏移位置映射到较高的内存地址中。当磁盘文件一旦被装入内存中，磁盘上的数据结构布局和内存中的数据结构布局是一致的。这样如果知道在磁盘的数据结构中寻找一些内容，那么几乎都能在被装入到内存映射文件中找到相同的信息。但数据之间的相对位置可能改变，其某项的偏移地址可能区别于原始的偏移位置，不管怎样，所有表现出来的信息都允许从磁盘文件偏移到内存偏移的转换（如图10.2所示）。

[image]

图10.2　PE文件磁盘与内存映像结构图

10.1.1　基地址

当PE文件通过Windows加载器被装入内存后，内存中的版本被称作模块（Module）。映射文件的起始地址被称为模块句柄（hModule），可以通过模块句柄访问内存中其他的数据结构。这个初始内存地址也称为基地址（ImageBase）。准确地说，对于Windows CE，这是不成立的，一个模块句柄在Windows CE下并不等同于安装的起始地址。

内存中的模块代表着进程从这个可执行文件中所需要的代码、数据、资源、输入表、输出表及其他有用的数据结构所使用的内存都放在一个连续的内存块中，编程人员只要知道装载程序文件映像到内存后的基地址即可。PE文件剩下的其他部分可以被读入，但是可能不映射。比如当调试信息放到文件尾部的时候，PE的一个字段会告诉系统把文件映

射到内存需要多少内存，不能被映射的数据将被放置在文件的尾部。

为了方便起见，Windows NT或Windows 95将Module的基地址作为Module的实例句柄（Instance Handle，即Hinstance）。在32位Windows系统中称基地址为Hinstance似乎有些混淆，因为Instance Handle一词来源于16位的Windows 3.1，其中每个执行实例都有自己的数据段，以此来互相区分，这就是Instance Handle的来历。在32位Windows系统中，已经不存在共享地址空间，所以已用程序无需加以区别。当然，16位Windows系统和32位Windows系统中的Hinstance还有些联系：在32位Windows系统中可以直接调用GetModuleHandle以取得指向DLL的指针，通过指针访问该DLL Module的内容。

[image]

当调用该函数时，传递一个可执行文件或DLL文件名字符串，如果系统找到文件，则返回该可执行文件或DLL文件映像加载到的基地址。也可调用GetModuleHandle，传递NULL参数，则返回调用的可执行文件的基地址。

基地址的值是由PE文件本身设定的。按照默认设置，用Visual C++建立的EXE文件基地址是00400000h，DLL文件基地址是10000000h。但是，可以在创建应用程序的EXE文件时改变这个地址，方法是在链接应用时使用链接程序的/BASE选项，或者链接后通过REBASE应用程序进行设置。

10.1.2　相对虚拟地址

在可执行文件中，有许多地方需要指定内存中的地址。例如，引用全局变量时，需要指定它的地址。PE文件尽管有一个首选的载入地址（基地址），但是它们可以载入到进程空间的任何地方，所以不能依赖于PE的载入点。由于这个原因，必须有一个方法来指定地址而不依赖于PE载入点的地址。

为了在PE文件中避免有确定的内存地址，出现了相对虚拟地址（Relative Virtual Address，简称RVA）概念。RVA只是内存中的一个简单的相对于PE文件装入地址的偏移位置，它是一个“相对”地址，或称为“偏移量”。例如，假设一个EXE文件从地址400000h处装入，并且它的代码区块开始于401000h，代码区块的RVA将是：

[image]

将一个RVA转换成真实的地址，只是简单地翻转这个过程：将实际的装入地址加上RVA即可得到实际的内存地址。顺便说一下，在PE用语里，实际的内存地址被称作虚拟地址（Virtual Address，简称VA），另外也可以把虚拟地址想象为加上首选装入地址的RVA。不要忘了前面提到的装入地址等同于模块句柄。它们之间的关系如下：

[image]

10.1.3　文件偏移地址

当PE文件储存在磁盘上时，某个数据的位置相对于文件头的偏移量，称为文件偏移地址（File Offset）或物理地址（RAW Offset）。文件偏移地址从PE文件的第一个字节开始计数，起始值为0。用十六进制工具（例如Hex Workshop、WinHex等）打开文件所显示的地址就是文件偏移地址。

10.2　MS-DOS头部

每个PE文件是以一个DOS程序开始的，有了它，一旦程序在DOS下执行，DOS就能识别出这是有效的执行体，然后运行紧随MZ header之后的DOS stub（DOS块）。DOS stub实际上是一个有效的EXE，在不支持PE文件格式的操作系统中，它将简单显示一个错误提示，类似于字符串“This program cannot be run in MS-DOS mode”。程序员也可根据自己的意图实现完整的DOS代码。用户通常对DOS stub不太感兴趣，因为大多数情况下它由汇编器/编译器自动生成。平常把DOS MZ头与DOS stub合称为DOS文件头。

PE文件的第一个字节起始于一个传统的MS-DOS头部，被称作IMAGE_DOS_HEADER。其IMAGE_DOS_HEADER结构如下所示（左边的数字是到文件头的偏移量）：

[image]

其中有两个字段比较重要，分别是e_magic和e_lfanew。e_magic字段（一个字大小）需要被设置为值5A4Dh，这个值有个#define，名为IMAGE_DOS_SIGNATURE，ASCII表示法里，它的ASCII值为“MZ”，是MS-DOS的最初创建者之一Mark Zbikowski字母的缩写。e_lfanew字段是真正PE文件头的相对偏移（RVA），其指出真正PE头的文件偏移位置，它占用4个字节，位于文件开始偏移3Ch字节中。

用十六进制编辑器（WinHex、Hex Workshop等带偏移量显示的尤佳）打开光盘映像文件上的PE.exe文件，定位在文件起始位置处，此处就是MS-DOS头部，如图10.3所示。文件第一个字符“MZ”就是e_magic字段。偏移3Ch就是e_lfanew的值，显示为“B0000000”。由于Intel CPU属于Little-Endian类，字符储存时低位在前，高位在后，将次序恢复后，e_lfanew的值为000000B0h，这个值就是真正的PE文件头偏移量。

[image]

图10.3　查看PE文件MS-DOS头部

10.3　PE文件头

紧跟着DOS stub的是PE文件头（PE Header）。PE Header是PE相关结构NT映像头（IMAGE_NT_HEADERS）的简称，其中包含许多PE装载器用到的重要字段。执行体在支持PE文件结构的操作系统中执行时，PE装载器将从IMAGE_DOS_HEADER结构中的e_lfanew字段里找到PE

Header的起始偏移量，加上基址得到PE文件头的指针。

[image]

实际上有两个版本的IMAGE_NT_HEADER结构，一个是为32位的可执行文件准备的，另一个是64位版本。在以后讨论中将不作考虑，它们几乎没有区别。

IMAGE_NT_HEADER是由三个字段组成（左边的数字是到PE文件头的偏移量）：

[image]

10.3.1　Signature字段

在一个有效的PE文件里，Signature字段被设置为00004550h，ASCII码字符是“PE00”，#define IMAGE_NT_SIGNATURE定义了这个值。

[image]

“PE\0\0”字串是PE文件头的开始，DOS头部的e_lfanew字段正是指向“PE\0\0”，如图10.3所示。

10.3.2　IMAGE_FILE_HEADER结构

IMAGE_FILE_HEADER（映像文件头）结构包含了PE文件的一些基本信息，最重要的是其中一个域指出了IMAGE_OPTIONAL_HEADER的大小。下面介绍IMAGE_FILE_HEADER结构的各个字段以及对这些字段的额外说明，这个结构也能在COFF格式的OBJ文件的最开始处找到，因此也称为COFF File Header。字段前的注释标出了字段相对于PE文件头的偏移量。

[image]

IMAGE_FILE_HEADER结构用十六进制工具显示情况如图10.4所示，图中的标号依次对应着相应的字段。

[image]

图10.4　IMAGE_FILE_HEADER结构

（1）Machine：可执行文件的目标CPU类型。PE文件可以在多种机器上使用，不同平台指令机器码是不同的。表10-1中所示是几种典型的机器类型标志。

表10-1　机器类型标志

[image]

（2）NumberOfSections：区块（Section）的数目，块表紧跟在IMAGE_NT_HEADERS后面。

（3）TimeDateStamp：表明文件是何时被创建的。这个值是自1970年1月1日以来用格林威治时间（GMT）计算的秒数，这个值是一个比文件系统的日期/时间更精确的文件创建时间指示器。将这个值翻译为易读的字符串的方法是用_ctime函数（它是时区敏感型的），另一个对此字段计算有用的函数是gmtime。

（4）PointerToSymbolTable：COFF符号表的文件偏移位置，描述于Microsoft规范的第5.4节。COFF符号表在PE文件中较少见，因为已采用了较新的debug格式。在Visual Studio .NET之前，COFF符号表可以通过设置链接器开关/DEBUGTYPE:COFF来创建。COFF符号表几乎总能在目标文件中找到，如果没有符号表存在，将此值设为0。

（5）NumberOfSymbols：如果有COFF符号表，它代表其中的符号数目，COFF符号是一个大小固定的结构，如果想找到COFF符号表的结束处，这个域是需要的。

（6）SizeOfOptionalHeader：紧跟着IMAGE_FILE_HEADER后面的数据大小。在PE文件中，这个数据结构叫IMAGE_OPTIONAL_HEADER，其大小依赖于是32位还是64位文件。对于32位PE文件，这个域通常是00E0h；对于64位PE32+文件，这个域是00F0h。不管怎么样，这些是要求的最小值，较大的值可能也会出现。

（7）Characteristics：文件属性，有选择地通过几个值的运算得到，这些标志的有效值是定义于winnt.h内的IMAGE_FILE_xxx值，具体值见表10-2。普通的EXE文件这个字段的值一般是010fh，DLL文件这个字段的值一般是210Eh。

表**10-2**　属性位字段的含义

[image]

10.3.3　IMAGE_OPTIONAL_HEADER结构

可选映像头（IMAGE_OPTIONAL_HEADER）是一个可选的结构，但实际上IMAGE_FILE_HEADER结构不足以定义PE文件属性，因此可选映像头中定义了更多的数据，完全不必考虑两个结构区别在哪里，两者连起来就是一个完整的“PE文件头结构”。IMAGE_OPTIONAL_HEADER32结构如下，字段前的注释标出了字段相对于PE文件头的偏移量。

[image]

[image]

IMAGE_OPTIONAL_HEADER32结构用十六进制工具显示情况如图10.5所示，图中的标号依次对应着相应的字段。

[image]

图10.5　IMAGE_OPTIONAL_HEADER32结构

（1）Magic：是一个标记字，说明文件是ROM映像（0107h），还是普通可执行的映像（010Bh），一般是010Bh，如是PE32+，则是020Bh。

（2）MajorLinkerVersion：链接程序的主版本号。

（3）MinorLinkerVersion：链接程序的次版本号。

（4）SizeOfCode：所有带有IMAGE_SCN_CNT_CODE属性区块的总共大小（只入不舍），这个值是向上对齐某一个值的整数倍。例如，本例是200h，即对齐的是一个磁盘扇区字节数（200h）的整数倍。通常情况下，多数文件只有一个Code块，所以这个字段和.text块的大小匹配。

（5）SizeOfInitializedData：已初始化数据块的大小，即在编译时所构成的块的大小（不包括代码段）。但这个数据并不太准确。

（6）SizeOfUninitializedData：未初始化数据块的大小，装载程序要在虚拟地址空间中为这些数据约定空间。这些块在磁盘文件中不占空间，就像“UninitializedData”这一术语所暗示的一样，这些块在程序开始运行时没有指定值。未初始化数据通常在.bss块中。

（7）AddressOfEntryPoint：程序执行入口RVA。对于DLL，这个入口点是在进程初始化和关闭时以及线程创建/毁灭时被调用。在大多数可执行文件中，这个地址并不直接指向Main、WinMain或DllMain，而是指向运行时库代码并由它来调用上述的函数。在DLL中这个域能被设置为0，前面提到的通知消息都不能收到。链接器/NOENTRY开关可以设置这个域为0。

（8）BaseOfCode：代码段的起始RVA。在内存中，代码段通常在PE文件头之后、数据块之前。在Microsoft链接器生成的执行文件中，RVA通常是1000h。Borland的Tlink32是将ImageBase加上第一个Code Section的RVA，并将结果存入该字段。

（9）BaseOfData：数据段的起始RVA。数据段通常是在内存的末尾，即PE文件头和Code Section之后。可是，这个域的值对于不同版本的微软链接器是不一致的，在64位可执行文件中是不出现的。

（10）ImageBase：文件在内存中的首选装入地址。如果有可能（也就是说，目前如果没有其他占据这块地址，它是正确对齐的并且是一个合法的地址，等等），加载器试图在这个地址装入PE文件。如果可执行文件是在这个地址装入的，那么加载器将跳过应用基址重定位的步骤。

（11）SectionAlignment：当被装入内存时的区块对齐大小。每个区块被装入的地址必定是本字段指定数值的整数倍。默认的对齐尺寸是目标CPU的页尺寸。对于运行在Windows 9x/Me下的用户模式可执行文件，最小的对齐尺寸是一页1000h（4KB）。这个字段可以通过链接器的/ALIGN开关来设置。在IA-64上，是按8KB来排列的。

（12）FileAlignment：磁盘上PE文件内的区块对齐大小，组成块的原始数据必须保证从本字段的倍数地址开始。对于x86可执行文件，

这个值通常是200h或1000h，这是为了保证块总是从磁盘的扇区开始，这个字段的功能等价于NE格式文件中的段/资源对齐因子。用不同版本的微软链接器默认值会改变。这个值必须是2的幂，其最小值为200h，并且如果SectionAlignment小于CPU的页尺寸，这个域必须与SectionAlignment匹配。链接器开关/OPT:WIN98设置x86可执行文件的文件对齐为1000h，/OPT:NOWIN98设置对齐为200h。

（13）MajorOperatingSystemVersion：要求操作系统的最低版本号的主版本号。随着这么多版本的Windows的到来，这个字段明显地变得不切题了。

（14）MinorOperatingSystemVersion：要求操作系统的最低版本号的次版本号。

（15）MajorImageVersion：该可执行文件的主版本号，由程序员定义。它不被系统使用并可以设置为0，可以通过链接器的/VERSION开关设置它。

（16）MinorImageVersion：该可执行文件的次版本号，由程序员定义。

（17）MajorSubsystemVersion：要求最低子系统版本的主版本号。这个值与下一个字段一起，通常被设置为4，可以通过链接器开关/SUBSYSTEM来设置。

（18）MinorSubsystemVersion：要求最低子系统版本的次版本号。

（19）Win32VersionValue：另一个从来不用的字段，通常被设置为0。

（20）SizeOfImage：映像装入内存后的总尺寸。它指装入文件从Image Base到最后一个块的大小。最后一个块根据其大小往上取整。

（21）SizeOfHeaders：是MS-DOS头部、PE头部、区块表的组合尺寸。所有这些项目都出现在PE文件中任何代码或数据区块之前。域值四舍五入至文件对齐的倍数。

（22）CheckSum：映像的校验和。IMAGEHLP.DLL中的CheckSumMappedFile函数可以计算这个值。一般的EXE文件可以是0，但一些内核模式的驱动程序和系统DLL必须有一个检验和。当链接器的/RELEASE开关被使用时，校验和被置于文件中。

（23）Subsystem：一个标明可执行文件所期望的子系统（用户界面类型）的枚举值。这个值只对EXE是重要的，具体见表10-3。

表**10-3**　界面子系统含义

[image]

（24）DllCharacteristics：DllMain()函数何时被调用，默认为0。

（25）SizeOfStackReserve：在EXE文件里，为线程保留的堆栈大小。它一开始只提交其中一部分，只有在必要时，才提交剩下的部分。

（26）SizeOfStackCommit：在EXE文件里，一开始即被委派给堆栈的内存数量。默认值是4KB。

（27）SizeOfHeapReserve：在EXE文件里，为进程的默认堆保留

的内存。默认值是1MB，但是在当前版本的Windows里，堆值在用户不干涉的情况下就能增长超过这个值。

（28）SizeOfHeapCommit：在EXE文件里，委派给堆的内存大小。默认值是4KB。

（29）LoaderFlags：与调试有关，默认为0。

（30）NumberOfRvaAndSizes：数据目录的项数。这个字段从最早的Windows NT发布以来一直是16。

（31）DataDirectory[16]：数据目录表，由数个相同的IMAGE_DATA_DIRECTORY结构组成，指向输出表、输入表、资源块等数据。IMAGE_DATA_DIRECTORY的结构定义如下：

[image]

数据目录表成员的结构如表10-4所示，各项成员含义后面会介绍。

表**10-4**　数据目录表成员

[image]

PE文件中定位输出表、输入表和资源等重要数据时，就是从IMAGE_DATA_DIRECTORY结构开始的。

本例数据目录表位于128h～1A7h之间，每个成员占8个字节，分别指向相关的结构，如图10.6所示。

[image]

图10.6　数据目录表

在图10.6中，地址128h就是数据目录表的第一项，其值为0，即这个实例的输出表地址与大小皆为0，表示无输出表。地址130h是第二项，该组数据表示输入表地址为2040h（RVA），大小为3Ch。

用PE编辑工具（如LordPE）来查看实例PE.exe文件的PE结构。单击LordPE的“PE Editor”打开PE_Offset文件，面板上直接显示出PE结构中的主要字段，如图10.7所示。

[image]

图10.7　用LordPE查看文件PE信息

然后单击“Directories”按钮，打开数据目录表查看面板，如图10.8所示。

[image]

图10.8　用LordPE查看PE文件数据目录表结构

10.4　区块

在PE文件头与原始数据之间存在一个区块表（Section Table），区块表包含每个块在映像中的信息，分别指向不同的区块实体。

10.4.1　区块表

紧跟着IMAGE_NT_HEADERS后的是区块表，它是一个IMAGE_SECTION_HEADER结构数组。每个IMAGE_SECTION_HEADER结构包含了它所关联区块的信息，如位置、长度、属性；该数组的数目由IMAGE_NT_HEADERS.FileHeader.NumberOfSections指出。

IMAGE_SECTION_HEADER结构定义如下：

[image]

为了方便理解，结合实例分析一下，实例PE.exe的区块表含有3个块的描述：.text、.rdata和.data。每个块对应一个IMAGE_SECTION_HEADER结构，用十六进制工具查看块表如图10.9所示。图中的标号依次对应着第一个IMAGE_SECTION_HEADER结构中的相应字段。

[image]

图10.9　十六进制工具中的块表

在图10.7中单击“Sections”按钮，打开区块编辑器（见图10.10），这是图10.9内容的另一种表现形式。

[image]

图10.10　LordPE查看的块表

（1）Name：块名。这是一个8位ASCII码名（不是Unicode内码），用来定义块名。多数块名以一个“.”开始（如.text），这个“.”实际上不是必需的。值得注意的是，如果块名超过8个字节，则没有最后的终止标志“NULL”字节。带有一个“$”的区块名字会从链接器那里得到特殊的对待，前面带有“$”的相同名字的区块被合并，在合并后的区块中，它们是按“$”后面的字符字母顺序进行合并的。

（2）VirtualSize：指出实际的、被使用的区块大小，是区块在没对齐处理前的实际大小。如果VirtualSize大于SizeOfRawData，那么SizeOfRawData是来自可执行文件初始化数据的大小，与VirtualSize相差的字节用零填充。这个字段在OBJ文件中是被设为0的。

（3）VirtualAddress：该块装载到内存中的RVA。这个地址是按照内存页对齐的，它的数值是SectionAlignment的整数倍。在Microsoft工具中，第一个块的默认RVA为1000h。在OBJ中，该字段没有意义，并被设为0。

（4）SizeOfRawData：该块在磁盘文件中所占的大小。在可执行文件中，该字段包含经过FileAlignment调整后的块的长度。例如，指定FileAlignment的大小为200h，如果VirtualSize中的块长度为19Ah个字

节，这一块应保存的长度为200h个字节。

（5）PointerToRawData：该块在磁盘文件中的偏移。程序经编译或汇编后生成原始数据，这个字段用于给出原始数据在文件中的偏移。如果程序自装载PE或COFF文件（而不是由操作系统装入），这一字段比VirtualAddress还重要。在这种状态下，必须完全使用线性映像方法装入文件，所以需要在该偏移处找到块的数据，而不是VirtualAddress字段中的RVA地址。

（6）PointerToRelocations：这部分在EXE文件中无意义。在OBJ文件中，表示本块重定位信息的偏移值。在OBJ文件中如果不是零，它会指向一个IMAGE_RELOCATION结构数组。

（7）PointerToLinenumbers：行号表在文件中的偏移值。这是文件的调试信息。

（8）NumberOfRelocations：这部分在EXE文件中无意义。在OBJ文件中，是本块在重定位表中的重定位数目。

（9）NumberOfLinenumbers：该块在行号表中的行号数目。

（10）Characteristics：块属性。该字段是一组指出块属性（如代码/数据/可读/可写等）的标志。比较重要的标志如表10-5所示，多个标志值求或即为Characteristics的值。这些标志中的很多都可以通过链接器的/SECTION选项设置。

表**10-5**　字段属性

[image]

例如，E0000020h = 20000000h | 40000000h | 80000000h | 00000020h表示该块包含执行代码，可读、可写并可执行。C0000040h = 40000000h | 80000000h | 00000040h表示该块可读、可写，包含已初始化的数据。60000020h = 20000000h | 40000000h | 00000020h表示该块包含执行代码，可读并可执行。

10.4.2　各种区块的描述

通常，区块中的数据逻辑上是关联的。PE文件一般至少有两个区块：一个是代码块，另一个是数据块。每一个区块都有一个截然不同的名字，这个名字是用来传达区块的用途。例如，一个区块叫.rdata，表明它是一个只读区块。区块在映像中是按起始地址（RVA）来排列的，而不是按字母表顺序。使用区块名字只是人们为了方便，而对操作系统来说是无关紧要的。微软给这些区块取了个有特色的名字，但这不是必需的。Borland的链接器用的是CODE和DATA这样的名字。

现在看看EXE和OBJ文件的一些常见区块（见表10-6），除非另外声明，表中的区块名称来自于微软定义。

表**10-6**　区块名称

[image]

[image]

[image]注意：当编程从PE文件中读取需要的内容时，如输入表、输出表等，不能以区块名称作为参考，正确的方法是按照数据目录表中的字段进行定位。

虽然编译器自动产生一系列标准的区块，但这没有什么不可思议。读者可以创建和命名自己的区块。在Visual C＋＋中，用#pragma来声明，告诉编译器插入数据到一个区块内，像下面这样：

[image]

这样所有被Visual C＋＋处理的数据都将放到一个叫MY_DATA的区块内，而不是默认的.data区块内。大部分的程序都只使用编译器产生的默认区块，但读者偶尔可能也会有一些特殊的需求，需要将代码或数据放到一个单独的区块里，例如建立一个全局共享块。

区块并不全部是在链接时形成的，更准确地说，它们一般是从OBJ文件开始，被编译器放置的。链接器的工作就是合并所有OBJ和库中需要的块，使其成为一个最终合适的区块。例如，在工程中的每一个OBJ至少都有一个包含代码的.text区块，链接器把这些区块合并成单一的.text区块。链接器遵循一套相当完整的规则，它判断哪些区块被合并以及如何被合并。OBJ文件中的一个区块可能是为链接器而准备的，不会放入最后的可执行文件中，像这样的区块主要用于编译器向链接器传递信息。

链接器一个有趣的特征就是能够合并区块。如果两个区块有相似、一致的属性，那么它们在链接时能合并成一个单一区块。这取决于是否用/merge开关。例如，下面的链接器选项将.rdata与.text区块合并为一个.text的区块。

[image]

合并区块的优点是可以节省磁盘和内存的空间。每个区块至少占用一个内存页，如果能将可执行文件内的区块数从4个减少到3个，很可能少用一页内存。当然，这依赖于两个合并区块的结尾未用空间加起来是否能达到一页。

当合并区块时，事情将变得有趣，因为这没有什么硬性规定。例如，把.rdata合并到.text里不会有什么问题，但是不应该将.rsrc、.reloc或.pdata合并到其他的区块里。在Visual Studio .NET之前，能将.idata合并到其他的区块里。在Visual Studio .NET中，这是不允许的。不过，当制作发行版本时，链接器经常将.idata的一部分合并到其他的区块里，如.rdata。

既然部分输入数据是在被装入内存时由Windows加载器写入的，那么读者可能对它们是如何被放入到只读区块表示疑惑。这种情况的发生是由于在加载时，系统会临时改变那些包含输入数据的页属性为可读、可写，初始化完成后，又恢复原来的属性。

10.4.3　区块的对齐值

区块的大小是要对齐的，有两种对齐值，一种用于磁盘文件内，另一种用于内存中。PE文件头指出了这两个值，它们可以不同。

PE文件头里FileAlignment定义了磁盘区块的对齐值。每一个区块从对齐值的倍数的偏移位置开始。而区块的实际代码或数据的大小不一定刚好是这么多，所以在不足的地方一般以00h来填充，这就是区块间的间隙。例如，在PE文件中，一个典型的对齐值是200h，这样，每个区块从200h之倍数的文件偏移位置开始，假设区块的第一个节在400h处，长度为90h，那么从文件400h到490h为这一区块的内容，而文件对齐值是200h，所以为了使这一节长度为FileAlignment的整数倍，490h到600h会被用零填充，这段空间称为区块间隙，下一个区块的开始地址为600h。

PE文件头里SectionAlignment定义了内存中区块的对齐值。PE文件被映射到内存中时，区块总是至少从一个页边界处开始，也就是说，当一个PE文件被映射到内存中，每个区块的第一个字节对应于某个内存页。在x86系列CPU中，页是按4KB（1000h）来排列的；在IA-64上，是按8KB（2000h）来排列的。所以在x86系统中，PE文件区块的内存对齐值一般等于1000h，每个区块按1000h之倍数的内存偏移位置开始。

再来回顾一下图10.10，.text区块在磁盘文件中的偏移位置是400h，在内存中将是其装入地址之上的1000h字节处。同样，.rdata区块在磁盘文件偏移的600h处，在内存中将是装入地址之上的2000h字节处。

Visual Studio 6.0中的默认值是4KB，除非使用/OPT:NOWIN98或/ALIGN开关。Visual Studio .NET的链接器，依然用了默认的/OPT:WIN98，但是如果文件小于某一特定大小时，就会采用200h为对齐值。另一种对齐方式来自于.NET文件的规定，它规定.NET文件的内存对齐值为8KB而不是普通x86平台上的4KB，这样就保证了在x86平台编译的程序可以在IA-64平台上运行。如果内存对齐值为4KB，那么IA-64加载器就不能载入这个程序，因为64位Windows中的页是8KB。

建立一个区块在文件中的偏移和在内存中的偏移相同的PE文件是可能的，这会使执行文件变大，但在Windows 9x/Me下可以加快装入速度。Visual Studio 6.0的默认选项/OPT:WIN98将会使PE文件按照这种方式来创建。在Visual Studio .NET中，链接器可以不使用/OPT:NOWIN98，这依赖于文件是否足够小。

10.4.4　文件偏移与虚拟地址转换

由于一些PE文件为减少体积，磁盘对齐值不是一个内存页1000h，而是200h，当这类文件被映射到内存后，同一数据相对于文件头的偏移量在内存中和磁盘文件中是不同的，这样就存在着文件偏移地址与虚拟地址的转换问题。而那些磁盘对齐值（1000h）与内存页相同的区块，同一数据在磁盘文件中的偏移和在内存中的偏移相同，不需要转换。

图10.10中区块显示出实例文件在磁盘与内存中各区块的地址、大小等信息。虚拟地址和虚拟大小是指该区块在内存中的地址和大小。物理地址和物理大小是指该区块在磁盘文件中的地址和大小。由于其磁盘对齐值为200h，与内存对齐值不同，故其磁盘映像和内存映像是不同

的，如图10.11所示。

[image]

图10.11　应用程序加载映射示意图

从图10.11中可看出，文件被映射到内存中，DOS文件头、PE文件头和块表的偏移位置与大小均没有变化。而各区块映射到内存后，其偏移位置就发生变化了。例如，磁盘文件中.text块起始端与文件头的偏移量为add1，映射到内存后，.text起始端与文件头（基地址）的偏移量为add2。同时，.text区块与块表之间形成一大段空隙，这部分数据全是以0填充的。这里，add1的值就是文件偏移地址（File Offset），add2的值就是相对虚拟地址（RVA）。假设它们的差值为Δk，则文件偏移地址与虚拟地址关系如下：

[image]

在同一区块中，各地址的偏移量是相等的，可用上面的公式对此区块中任一File Offset与VA进行转换。但请不要错误地认为在整个文件里，File Offset与VA的差值是Δk。因为各区块在内存中是以一个页边界为开始的，第一个区块结束后，一直到第二个区块起始端（1000h对齐处）全以数据0填充，所以不同区块在磁盘与内存中的差值不一样。表10-7所示是该实例文件各区块在磁盘与内存中的起始地址差值。

表**10-7**　各区块在磁盘与内存中的起始地址差值

[image]

例如，此实例中某一虚拟地址（VA）= 401112h，要求计算它的文件偏移地址。401112h在.text块中，此时Δk = 0C00h，故

[image]

再来看一看虚拟地址4020D2h的转换：

4020D2h是在.rdata块中，此时Δk = 1A00h，故

[image]

在实际操作时，建议使用RVA-Offset之类的转换工具。LordPE工具也有这个转换功能，单击图10.7中的“FLC”按钮打开“文件位址计算器（File Location Calculator）”，见图10.12。

[image]

图10.12　地址转换器

在VA域中输入要转换的虚拟地址401112h，单击“DO”按钮，转换后的文件偏移地址为512h。

10.5 输入表

可执行文件使用来自于其他DLL的代码或数据时，称为输入。当PE文件装入时，Windows加载器的工作之一就是定位所有被输入的函数和数据，并且让正在被装入的文件可以使用那些地址。这个过程是通过PE文件的输入表（Import Table，简称IT，也称为导入表）来完成的，输入表中保存的是函数名和其驻留的DLL名等动态链接所需的信息。输入表在软件外壳技术上的地位非常重要，因此读者在研究外壳相关技术时，一定要彻底掌握这部分知识。

10.5.1 输入函数的调用

在代码分析或编程中经常遇到“输入函数（Import functions）”。输入函数就是被程序调用但其执行代码又不在程序中的函数，这些函数的代码位于相关的DLL文件中，在调用者程序中只保留相关函数信息，如函数名、DLL文件名等。对于磁盘上的PE文件来说，它无法得知这些输入函数在内存中的地址。只有当PE文件被装入内存后，Windows加载器才将相关DLL装入，并将调用输入函数的指令和函数实际所处的地址联系起来。

当应用程序调用一个DLL的代码和数据时，那它正在隐含链接到DLL，这个过程完全由Windows加载器完成。另一种是运行期的显式链接，这意味着必须确定目标DLL已经被加载，然后寻找API的地址，这几乎总是通过调用LoadLibrary和GetProcAddress来完成的。

当隐含地链接一个API时，类似LoadLibrary和GetProcAddress的代码始终在执行，只不过这是Windows装载器自动完成的。装载器还保证PE文件所需的任何附加的DLL都已被载入。例如，Windows 2000/XP上每个由Visual C＋＋创建的正常程序都要链接KERNEL32.DLL，而它又从NTDLL.DLL输入函数。同样地，如果链接了GDI32.DLL，它又依赖于USER32、ADVAPI32、NTDLL和KERNEL32等DLL的函数，这些都是由加载器来保证装入并解决输入问题。

在PE文件内，有一组数据结构，它们分别对应着每个被输入的DLL。每一个这样的结构都给出了被输入的DLL的名称并指向一组函数指针。这组函数指针被称为输入地址表（Import Address Table，简称IAT）。每一个被引入的API在IAT里都有它自己保留的位置，在那里它将被Windows加载器写入输入函数的地址。最后一点是特别重要的：一旦模块被装入，IAT中包含所要调用输入函数的地址。

把所有输入函数放在IAT中同一个地方是很有意义的，这样无论代码中多少次调用一个输入函数，都会通过IAT中的同一个函数指针来完成。

现在看看怎样调用一个输入函数。需要考虑两种情况：高效和低效。最好的情况是像下面这样：

[image]

直接调用[00402010]中的函数，地址402010h位于IAT里。而实际上对一个被输入的API低效的调用像下面这样（下面是实例PE.exe中的

调用LoadIconA函数的代码）：

[image]

这种情况，CALL把控制权转到一个子程序，子程序中的JMP指令跳转到位于IAT中的402010h。简单地说，它使用5个字节的额外代码，并且由于额外的JMP将花费更多的时间去执行。

读者可能会问，为什么还采用此种低效方法？有个很好的解释，编译器无法区别输入函数的调用和普通函数调用。对于每个函数调用，编译器使用同样形式的CALL指令：

[image]

XXXXXXXX是一个由链接器填充的实际的地址。注意，这条指令不是从函数指针而是代码中的实际地址而来的。为了因果的平衡，链接器必须产生一块代码来取代XXXXXXXX，简单的方法就是像上面所示调用一个JMP stub。

JMP指令来自于为输入函数准备的输入库。如果读者曾经检查过某个输入库，在输入函数名字的关联处就会发现与上面JMP stub相似的指令。这意味着在默认情况下，对被输入API的调用将使用低效的形式。

如何得到优化的形式？答案来自于给编译器的一个提示形式。可以用修饰函数的__declspec(dllimport)来告诉编译器，这个函数来自另一个DLL中，这样编译器就会产生这样的指令：

[image]

而不是：

[image]

此外，编译器将给函数加上__imp_前缀，然后送给链接器，这样可以直接把__imp_xxx送到IAT，就不需要JMP stub了。

如果在写一个输出函数并且为它们提供一个头文件的话，别忘了在函数前加上修饰符__declspec(dllimport)，在winnt.h等系统头文件中就是这样做的。

[image]

10.5.2　输入表结构

PE文件头的可选映像头中数据目录表的第二成员指向输入表。输入表以一个IMAGE_IMPORT_DESCRIPTOR（简称IID）数组开始。每个被PE文件隐式地链接进来的DLL都有一个IID。在这个数组中，没有字段指出该结构数组的项数，但它的最后一个单元是NULL，可以由此计算出该数组的项数。例如，某个PE文件从两个DLL文件中引入函数，就存在两个IID结构来描述这些DLL文件，并在两个IID结构的最后由一个内容全为0的IID结构作为结束。

IID的结构如下：

[image]

• OriginalFirstThunk（Characteristics）：包含指向输入名称表（简称INT）的RVA，INT是一个IMAGE_THUNK_DATA结构的数组，数组中的每个IMAGE_THUNK_DATA结构指向IMAGE_IMPORT_BY_NAME结构，数组最后以一个内容为0的IMAGE_THUNK_DATA结构结束。

• TimeDateStamp：一个32位的时间标志，可以忽略。

• ForwarderChain：这是第一个被转向的API的索引，一般为0。当程序引用一个DLL中的API，而这个API又引用别的DLL的API时使用，但这样的情况很少出现。

• Name：DLL名字的指针。是个以00结尾的ASCII字符的RVA地址，该字符串包含输入的DLL名，例如“KERNEL32.DLL”或“USER32.DLL”。

• FirstThunk：包含指向输入地址表（IAT）的RVA。IAT是一个IMAGE_THUNK_DATA结构的数组。

OriginalFirstThunk与FirstThunk非常相似，它们指向两个本质上相同的数组IMAGE_THUNK_DATA，这些数组有好几种叫法，但最常见的名字是输入名称表（Import Name Table, INT）和输入地址表（Import Address Table, IAT）。图10.13表示了一个可执行文件正在从USER32.DLL里输入一些API。

[image]

图10.13　两个并行的指针数组

两个数组都有IMAGE_THUNK_DATA结构类型的元素，它是一个指针大小的联合（union）。每一个IMAGE_THUNK_DATA元素对应于一个从可执行文件输入的函数。两个数组的结束是通过一个值为零的IMAGE_THUNK_DATA元素来表示的。IMAGE_THUNK_DATA结构实际上是一个双字，该结构在不同时刻有不同的含义。定义如下：

[image]

当IMAGE_THUNK_DATA值的最高位为1时，表示函数以序号方式输入，这时低31位（或者一个64位可执行文件的低63位）被看作是一个函数序号。当双字的最高位为0时，表示函数以字符串类型的函数名方式输入，这时双字的值是一个RVA，指向一个IMAGE_IMPORT_BY_NAME结构。

IMAGE_IMPORT_BY_NAME结构仅仅是一个字大小，存有一个输入函数的相关信息结构。其结构如下：

[image]

• Hint：指示本函数在其所驻留DLL的输出表中的序号。该域被PE装载器用来在DLL的输出表里快速查询函数。该值不是必需的，一些链

接器将此值设为0。

• Name：含有输入函数的函数名，函数名是一个ASCII码字符串，以NULL结尾。注意，这里虽然将Name的大小定义成字节，其实它是可变尺寸域，由于没有更好的表示方法，只好在上面定义中写成BYTE。

10.5.3　输入地址表（IAT）

为什么由两个并行的指针数组指向IMAGE_IMPORT_BY_NAME结构呢？第一个数组（由OriginalFirstThunk所指向）是单独的一项，而且不可改写，称为INT，有时也称为提示名表（Hint-name Table）。第二个数组（由FirstThunk所指向）是由PE装载器重写的。PE装载器首先搜索OriginalFirstThunk，如果找到，加载程序迭代搜索数组中的每个指针，找到每个IMAGE_IMPORT_BY_NAME结构所指向的输入函数的地址，然后加载器用函数真正入口地址来替代由FirstThunk指向的IMAGE_THUNK_DATA数组里的元素值。Jmpdword ptr [xxxxxxxx]中的[xxxxxxxx]是指First Thunk数组中的一个入口，因此它称为输入地址表（Import Address Table, IAT）。因此，当PE文件装载内存后准备执行时，图10.13已转换成图10.14所示的状态，所有函数入口地址被排列在一起。此时，输入表中其他部分就不重要了，程序依靠IAT提供的函数地址就可正常运行。

[image]

图10.14　PE文件加载后的IAT表

有些情况下，一些函数仅由序号引出，也就是说，不能用函数名来调用它们，只能用它们的位置来调用。此时，IMAGE_THUNK_DATA值的低位字指示函数序数，而最高二进位（MSB）设为1。Microsoft提供了一个方便的常量来测试DWORD值的MSB位，就是IMAGE_ORDINAL_FLAG32，其值为80000000h（PE32+中是IMAGE_ORDINAL_FLAG64，其值为8000000000000000h）。

另外一种情况是程序OrignalFirstThunk的值为0。在初始化时，系统根据FirstThunk的值找到指向函数名的地址串，由地址串找到函数名，再根据函数名得到入口地址，然后用入口地址取代FirstThunk指向的地址串中的原值。

10.5.4　输入表实例分析

下面就来分析实例PE.exe文件的输入表。数据目录表的第二成员指向输入表，该指针具体位置是在PE文件头的80h偏移处。该文件的PE文件头起始位置是B0h，输入表地址就是在整个文件的B0h + 80h = 130h处，因此在130h处可以发现四字节指针40 20 00 00，倒过来就是00002040，即输入表在内存中偏移量为2040h的地方。当然，这个2040h是RVA值，需要将其转换为磁盘文件的绝对偏移量，才能够在十六进制编辑器中找到输入表。

可以用LordPE之类的PE编辑工具来查看各个块的实际偏移，以便确定2040h到底指的是什么地方。为了加强理解，在此手动转换，从图

10.10可知，2040h位于.rdata块中，.rdata块的虚拟偏移是2000h，其物理偏移是600h，因此Δk = 2000h - 600h = 1A00h，这个数字在后面的两种偏移量转换中需要用到，现在先记住它。相对虚拟地址（RVA）2040h转换成文件偏移地址：2040h - 1A00h = 640h。

用十六进制工具打开文件，跳到偏移640h处，这里就是输入表的内容，每个IID包含5个双字，用来描述一个引入的DLL文件，最后以NULL结束，图10.15列出了输入表的一部分。

[image]

图10.15　磁盘文件中的输入表

将图10.15所列的输入表的IID数组（图中阴影部分）整理到表10-8中。每个IID包含了一个DLL的描述信息，现在有两个IID，因此这里引入了两个DLL，第三个IID全为0，作为结束标志。

表**10-8**　十六进制工具中显示的**IID**数组

[image]

每个IID中的第四个字段是指向DLL名称的指针。这里第一个IID中的第四个字段是7421 0000，翻转过来也就是RVA地址00002174h，将它减去1A00h得到文件偏移地址774h，于是，查看EXE文件中偏移量为774h处的字符，是什么？原来调用的是USER32.dll。经转换后的IID数组见表10-9，表内各指针都是RVA地址。

表**10-9**　**IID**数组

[image]

再找USER32.dll中被调用的函数。仍然在第一个IID中，查看第一个字段OrignalFirstThunk，它指向一个数组，这个数组的元素都是指针，分别指向引入函数名的ASCII字符串。有些程序的OriginalFirstThunk值为0，所以这时就要看FirstThunk，它在程序运行时被初始化。

USER32.dll所在IID的OrignalFirstThunk字段值是208Ch，减去1A00h得68Ch，在偏移68Ch处就是IMAGE_THUNK_DATA数组，它存储的是指向IMAGE_IMPORT_BY_NAME结构的地址，以一串00结束，如表10-10所示。

表**10-10**　**IMAGE_THUNK_DATA**数据

[image]

再来看看同一IID结构中FirstThunk情况，USER32.dll所在IID的FirstThunk字段值是2010h，减去1A00h得610h，在偏移610h处就是IMAGE_THUNK_DATA数组，其数据与OrignalFirstThunk字段所指的完全一样（见表10-10）。

通常，在一个完整的程序里都有这些。现在有11个IMAGE_THUNK_DATA，表示有11个函数调用，先看看其中的两个。

1021 0000翻转后是2110h，减去1A00h后等于710h，会发现在偏移710h处的字符串是LoadIconA。

1C21 0000翻转后是211Ch，减去1A00h后等于71Ch，会发现在偏移71Ch处的字符串是PostQuitMessage。

读者也许注意到了，计算出来的偏移量并不刚好指向函数名的ASCII字符串，而是前面还有两个字节的空缺，这是作为函数名（Hint）引用的，可以为0。

表10-11所示是第一个IID指向的各种API函数。

表10-11　第一个IID指向的API函数

[image]

图10.16是PE.exe文件运行前第一个IID的结构示意图。在程序运行前，它的FirstThunk字段值也是指向一个地址串，而且和OrignalFirstThunk字段值指向的INT是重复的。系统在程序初始化时根据OrignalFirstThunk的值找到函数名，调用GetProcAddress函数（或类似功能的系统代码）且根据函数名取得函数的入口地址，然后用函数入口地址取代FirstThunk指向的地址串中对应的值（IAT）。

[image]

图10.16　第一个IID在磁盘文件里的结构

实例dumped.exe是从内存中抓取出来的，因此其结构就是PE文件映射到内存的状态。打开映像文件，由于在内存中区块的对齐值与内存页相同，因此此时其文件偏移地址与相对虚拟地址（RVA）的值相等。输入表的RVA地址是2040h，具体见图10.17。

[image]

图10.17　内存中的部分输入表

从图10.17中看出，OrignalFirstThunk字段指向的数据没变，但FirstThunk字段指向的数据已改变（图中阴影部分为IAT）。第一个IID的FirstThunk字段为2010h，该RVA（2010h）指向输入地址表（IAT），将这张表的数据整理进表10-12（笔者当时的系统是Windows XP SP1，不同系统其值不同）。

表10-12　内存中第一个IID结构的输入地址表（IAT）

[image]

表10-12中各地址都是USER32.dll链接库的相关输出函数，先反汇编USER32.dll（反汇编技术参考静态分析一章），跳到77D216DDh地

址处，显示代码如下：

[image]

原来，77D216DD指向的是USER32.dll中LoadIconA函数代码处。图10.18是PE.exe文件装载到内存里的第一个IID的结构示意图。

[image]

图10.18　第一个IID装载到内存里的结构

程序装载进内存中后，只与IAT交换信息，输入表的其他部分不需要了。例如，程序调用LoadIconA函数的指针是指向IAT的，而IAT已指向系统USER32.dll的LoadIconA函数代码里。调用LoadIconA函数的相关代码如下：

[image]

这个程序有两个DLL，读者可参考上面的过程分析第二个IID。

10.6　绑定输入

当PE装载器装入PE文件时，检查输入表并将相关DLL映射到进程地址空间。然后遍历IAT里的IMAGE_THUNK_DATA数组并用输入函数的真实地址替换它，这一步需要很多时间。如果程序员事先能正确预测函数地址，PE装载器就不用每次装入PE文件时都去修正IMAGE_THUNK_DATA值了，绑定输入（Bound Import）就是这种思想的产物。

当一个可执行文件被绑定（例如通过绑定程序Visual Studio的Bind.exe）时，IAT中的IMAGE_THUNK_DATA结构被输入函数的实际地址改写了。磁盘中的可执行文件，它们的IAT里有的存放的是相关DLL输出函数的实际内存地址。这样可以使应用程序更快地进行初始化，并且使用较少的存储器。

在执行整个进程期间，Bind程序做了两个重要假设：

• 当进程初始化时，需要的DLL实际上加载到了它们的首选基地址中。

• 自从绑定操作执行以来，DLL输出表中引用的符号位置一直没有改变。

当然，如果上面的两个假设中有一个是假的，IAT中所有地址均是无效的，加载器会检查这种情况并做出相应反应，加载器从INT表里获得所需要的信息来解决输入API的地址问题。对于一个可执行文件的装入，INT是不需要的。但是，如果没有，可执行文件是不能被绑定的。微软的链接器似乎总是生成一个INT，但是在很长一段时间里，Borland的链接器（TLINK）没有这样做，由Borland生成的文件是不能被绑定的。

由于不知用户运行的是Windows 2000还是Windows XP，无法将系统DLL绑定起来，因此程序安装时是绑定程序的最佳时机。Windows安

装器的BindImage将做这些工作。另外，IMAGEHLP.DLL提供了BindImageEx函数。不管用什么方式，绑定都是一个好主意。如果加载器确定绑定信息是当前的，可执行文件的装入会更快，如果绑定信息已经变得陈旧了，也不会影响程序的运行。

对于加载器来说，使绑定变得有效的一个关键步骤是确定在IAT表中的绑定信息是否是当前的。当一个可执行文件被绑定时，被参考的DLL信息放入了文件中，加载器检查这些信息来做一个快速的绑定有效性验证。

数据目录表（DataDirectory）的第12个成员指向绑定输入。绑定输入以一个IMAGE_BOUND_IMPORT_DESCRIPTOR结构的数组开始，一个绑定可执行文件包含一系列这样的结构，每个IBID结构都指出了一个已经被绑定输入DLL的时间/日期戳。IBID的结构如下：

[image]

• TimeDateStamp：一个双字，包含一个被输入DLL的时间/日期戳；允许加载器快速判断绑定是不是新的。

• OffsetModuleName：一个字，包含一个指向被输入DLL的名称的偏移。这个字段是与第一个IBID结构之间的偏移（不是RVA）。

• NumberOfModuleForwarderRefs：一个字，它包含紧跟在这个结构后面的IMAGE_BOUND_FORWARDER_REF结构的数目。这些结构是和IBID相同的，除了最后的一个字（NumberOfModuleForwarderRef）被保留。

当绑定一个API被转向到另一个DLL时，被转向到的DLL的有效性也要被检查。这样，IMAGE_BOUND_FORWARDER_REF和IMAGE_BOUND_IMPORT_DESCRIPTOR结构是交叉存取的。

例如链接到HeapAlloc，它被转向到NTDLL中的RtlAllocateHeap，然后对可执行文件运行BIND。在EXE里，已经有一个针对KERNEL32.DLL的IBID，它后面跟着一个针对NTDLL.DLL的IMAGE_BOUND_FORWARDER_REF。紧跟在后面可能是另外的你输入并绑定的针对其他DLL的IBID。

Windows目录里的应用程序就是典型的绑定输入结构程序，其IAT已指向相关DLL的函数。图10.19是Windows XP记事本程序（Notepad.exe）的绑定输入结构，此时记事本程序的IAT全是指向了系统DLL相关函数的入口地址。

[image]

图10.19　绑定输入的结构

为了方便实现，Microsoft的一些编译器（如Visual Studio）都提供了bind.exe这样的工具，由它检查PE文件的输入表，并用输入函数的真实地址替换IAT里的IMAGE_THUNK_DATA值。当文件装入时，PE装载器必定检查地址的有效性。如果DLL版本不同于PE文件存放的相关信息，或DLL需要重定位，那么装载器认为原先计算的地址是无效的，它必定遍历OriginalFirstThunk指向的数组以获取输入函数新地址，产生一

个新的IAT。

绑定输入表去除是不会影响程序正常运行的，去除方法是将图10.19中的绑定数据清零，然后再将目录表中的Bound import的RVA与大小清零即可。

10.7 输出表

当创建一个DLL时，实际上创建了一组能让EXE或其他DLL调用的一组函数，此时PE装载器根据DLL文件中输出信息修正被执行文件的IAT。当一个DLL函数能被EXE或另一个DLL文件使用时，它被称为输出了（exported）。其中输出信息被保存在输出表中，DLL文件通过输出表向系统提供输出函数名、序号和入口地址等信息。

EXE文件一般不存在输出表，而大部分DLL文件中存在输出表。当然，这也不是绝对的，有些EXE文件也会存在输出函数。

10.7.1 输出表结构

输出表（Export Table）中的主要成分是一个表格，内含函数名称、输出序数等。序数是指定DLL中某个函数的16位数字，在所指向的DLL里是独一无二的。在此不提倡仅仅通过序数引出函数这种方法，这会带来DLL维护上的问题。一旦DLL升级或修改，调用该DLL的程序将无法工作。

输出表是数据目录表的第一个成员，指向IMAGE_EXPORT_DIRECTORY（简称IED）结构。IED结构定义如下：

[image]

说明：

• Characteristics：表示输出属性的旗标。目前还没有定义，总是为0。

• TimeDateStamp：输出表创建的时间（GMT时间）。

• MajorVersion：输出表的主版本号。未使用，设置为0。

• MinorVersion：输出表的次版本号。未使用，设置为0。

• Name：指向一个ASCII字符串的RVA，这个字符串是与这些输出函数关联的DLL的名字（例如KERNEL32.DLL）。

• Base：这个字段包含用于这个可执行文件输出表的起始序数值（基数）。正常地，这个值是1，但是并不需要非得这样。当通过序数来查询一个输出函数时，这个值从序数里被减去，结果被用作进入输出地址表（EAT）的索引。

• NumberOfFunctions：EAT中的条目数量。注意，一些条目可能是0，表明用这个序数值没有代码或数据被输出。

• NumberOfNames：输出函数名称表（ENT）里的条目数量。这个值总是小于或等于NumberOfFunctions域值。小于的情况发生在符号只通过序数来输出，另外当被赋值的序数里有数字间距时也会是小于的，这个值也是输出序数表的长度。

• AddressOfFunctions：EAT的RVA。EAT是一个RVA数组，数组中

的每一个非零的RVA都对应于一个被输出的符号。

• AddressOfNames：ENT的RVA。ENT是一个指向ASCII字符串的RVA数组。每一个ASCII字符串对应于一个通过名字输出的符号。这个表是排序的，所以ASCII字符串也是按顺序排列的。这允许加载器在查询一个被输出的符号时用二进制查找方式，名称的排序是二进制的（像C＋＋ RTL中strcmp函数提供的一样），而不是一个环境特定的字母顺序。

• AddressOfNameOrdinals：输出序数表的RVA。这个表是字的数组。这个表将ENT中的数组索引映射到相应的输出地址表条目。

输出表的设计是为了方便PE装载器工作。首先，模块必须保存所有输出函数的地址，供PE装载器查询。模块将这些信息保存在AddressOfFunctions域所指向的数组中，而数组元素数目存放在NumberOfFunctions域中。如果模块引出40个函数，则AddressOfFunctions指向的数组必定有40个元素，而NumberOfFunctions值为40。如果有些函数是通过名字引出的，那么模块必定也在文件中保留了这些信息。这些名字的RVA存放在一个数组中，供PE装载器查询。该数组由AddressOfNames指向，NumberOfNames包含名字数目。PE装载器知道函数名，并想以此获取这些函数的地址。至今为止，已有两个模块：名字数组和地址数组，但两者之间还没有联系的纽带，还需要一些联系函数名及其地址的东西。PE参考指出使用到地址数组的索引作为连接，因此PE装载器在名字数组中找到匹配名字的同时，它也获取指向地址表中对应元素的索引。这些索引保存在由AddressOfNameOrdinals域所指向的另一个数组（最后一个）中。由于该数组起到联系名字和地址的作用，所以其元素数目必定和名字数组相同。例如，每个名字有且仅有一个相关地址，反过来则不一定：每个地址可有好几个名字来对应。因此，给同一个地址取“别名”。为了起到连接作用，名字数组和索引数组必须并行成对使用，比如索引数组的第一个元素必定含有第一个名字的索引，依次类推。图10.20显示了Export Table的格式及其中的3个阵列。

[image]

图10.20　一个典型的输出表（Export Table）

10.7.2　输出表结构实例分析

在这里以光盘映像文件中提供的DllDemo.DLL这个实例了解输出表。数据目录表的第一个成员指向输出表，该指针具体位置是在PE文件头的78h偏移处。该文件的PE文件头起始位置是100h，输出表就是在整个文件的100h + 78h = 178h处，因此在178h处可以发现四字节指针00 40 00 00，倒过来就是00004000，即输出表在内存中偏移4000h的地方。当然，这个4000h指的是内存中的偏移量，转成文件偏移地址就是0C00h。文件偏移0C00h处是输出表内容，具体如图10.21所示。

[image]

图10.21　输出表

这个DLL只有一个输出函数MsgBox，其IMAGE_EXPORT_DIRECTORY结构如表10-13所示。

表10-13　IMAGE_EXPORT_DIRECTORY结构

[image]

（1）Name：4032h - 3400h = C32h，指向DLL名字DllDemo.DLL。

（2）AddressOfNames：402Ch - 3400h = C2Ch，指向函数名的指针403Eh - 3400h = C3Eh，C3Eh再指向函数名MsgBox。

（3）AddressOfNameOrdinals：4030h - 3400h = C30h，指向输出序号数组。

再来看看输出是如何实现的。PE装载器调用GetProcAddress来查找DllDemo.DLL里的API函数MsgBox，系统通过定位DllDemo.DLL的IMAGE_EXPORT_DIRECTORY结构开始工作，从这个结构中，它获得输出函数名称表（Export Names Table，简称ENT）起始地址，进而知道这个数组里一共有1个条目，它对名字进行二进制查找直到发现字符串“MsgBox”。

PE装载器发现MsgBox是数组的第一个条目，加载器然后从输出序数表读取相应的第一个值，这个值是MsgBox的输出序数。使用输出序数作为进入EAT的索引（并且也要考虑Base域值），它得到MsgBox的RVA是1008h，1008h加上DllDemo.DLL的装入地址得到MsgBox的实际地址。

10.8　基址重定位

当链接器生成一个PE文件时，它假设这个文件执行时会被装载到默认的基地址处，并且把code和data的相关地址都写入PE文件中。如果装入时按默认的值作为基地址装入，则不需要重定位。但如果可执行文件被装载到虚拟内存的另一个地址，链接器所登记的那个地址就是错误的，这时就需要用重定位表来调整。在PE文件中，它往往单独分为一块，用“.reloc”表示。

10.8.1　基址重定位概念

和NE格式的重定位方式不同，PE的做法十分简单。它们并不参考外部DLL或模块中的其他sections，而是把文件中所有可能需要修改的地址放在一个数组里。如果可执行文件不在首选的地址装入，那么文件中每一个定位都需要被修正。对加载器来说，它不需要知道关于地址如何使用的任何细节，它只需知道有一系列的数据需要以某种一致的方式来修正就可以了。

下面以实例DllDemo.DLL为例讲述其重定位过程。下面代码中两个加粗的地址指针是需要重定位的数据。

[image]

来分析一下0040100Eh这句，其作用将一个指针压进栈，402000h是某一字符串的指针。这句指令5字节长，前1个字节（68h）是指令的操作码，后4个字节用来保存一个DWORD大小的地址（00402000h）。在这个例子中，指令是来自一个基址为00400000h的DLL文件，因此这个字符串的RVA是2000h。

如果可执行文件确实在00400000h处装入，那么指令能够按照现在的样子正确执行。但是当DLL执行时，Windows加载器决定将其映射到870000h处（映射基址由系统决定）。加载器会比较基址和实际的装入地址，计算出一个差值。在这个例子中，差值是470000h。这个差值能被加到DWORD大小的地址值里以形成新地址。在前面的例子中，地址0040100Fh是指令中双字的定位，对它将有一个基址重定位，实际上字符串新的地址就是872000h。为了让Windows有能力这样调整，可执行文件中内含许多个“基址重定位数据”。本例中的装载器应把470000h加给402000h，并将872000h结果写回原处。图10.22演示了这个过程。

[image]

图10.22　PE文件重定位过程

DllDemo.DLL在内存中重定位处理过的代码如下：

[image]

对于EXE文件来说，每个文件总是使用独立的虚拟地址空间，所以EXE总是能够按照这个地址装入，这意味着EXE文件不再需要重定位信息。对于DLL来说，由于多个DLL文件全部使用宿主EXE文件的地址空间，不能保证装入地址没有被其他的DLL使用，所以DLL文件中必须包含重定位信息，除非用一个/FIXED开关来省略它们。在Visual Studio .NET中，链接器会为Debug和Release模式的EXE文件省略掉基址重定位。因此在不同系统上跟踪同一个DLL文件时，其虚拟地址都是不相同的。也就是说，在读者的机器里运行DllDemo.DLL，装载器映射的基址可能不是00870000h，而是其他的值。

10.8.2　基址重定位结构定义

基址重定位表（Base Relocation Table）位于一个叫.reloc的区块内，但是找到它们的正确方式是通过数据目录表的IMAGE_DIRECTORY_ENTRY_BASERELOC条目。基址重定位数据组织方法采用类似按页分割的方法，其由许多重定位块串接成的，每个块存放着4KB（1000h）大小的重定位信息，每个重定位数据块的大小必须以DWORD（4字节）对齐。它们以一个IMAGE_BASE_RELOCATION结构开始，格式如下：

[image]

• VirtualAddress：是这一组重定位数据的开始RVA地址。各重定位项的地址加上这个值才是该重定位项完整的RVA地址。

• SizeOfBlock：是当前重定位结构的大小。因为VirtualAddress和SizeOfBlock大小都是固定的4个字节，因此这个项减去8，则是TypeOffset大小。

• TypeOffset：是一个数组。数组每项大小为两个字节，共16位。它又分为高4位与低12位，高4位代表重定位类型；低12位是重定位地址，它与VirtualAddress相加即是指向PE映像中需要修改的地址数据的指针。

常见的重定位类型见表10-14。虽然有多种重定位类型，对于x86可执行文件，所有的基址重定位类型都是IMAGE_REL_BASED_HIGHLOW。在一组重定位结束的地方会出现一个类型是IMAGE_REL_BASED_ABSOLUTE的重定位，这些重定位什么都不做，在那里只用于填充，以便下一个IMAGE_BASE_RELOCATION是以4字节分界线来对齐。所有重定位块最终以一个VirtualAddress字段为0的IMAGE_BASE_RELOCATION结构作为结束。

表10-14　常见的重定位类型

[image]

重定位表的结构如图10.23所示，由数个IMAGE_BASE_RELOCATION结构组成，每个结构由VirtualAddress、SizeOfBlock和TypeOffset三部分组成。

[image]

图10.23　重定位表示意图

对于IA-64可执行文件，重定位似乎总是IMAGE_REL_BASED_DIR64类型。就像x86重定位，它们也用IMAGE_REL_BASED_ABSOLUTE重定位类型来进行填充。有趣的是，尽管IA-64的EXE页大小是8KB，但基址重定位仍旧是4KB的块。

10.8.3　基址重定位结构实例分析

下面以DllDemo.DLL为例来讲解。数据目录表指向重定位表的指针是5000h，换算成文件偏移地址就是0E00h。其IMAGE_BASE_RELOCATION结构如图10.24所示。

[image]

图10.24　基址重定位表

VirtualAddress：00001000h

SizeOfBlock：00000010h（有4个重定位数据，（10h - 8h）/2h = 4h）

- 重定位数据1：300Fh
- 重定位数据2：3023h
- 重定位数据3：0000（用于对齐）
- 重定位数据4：0000（用于对齐）

重定位数据计算过程见表10-15。

表10-15　重定位数据转换

[image]

用十六进制工具查看实例文件，其中060Fh和623h分别指向402000h和403030h，图10.25阴影部分即为所需要重定位的数据。

[image]

图10.25　需要重定位的数据

执行PE文件前，加载程序在进行重定位的时候，会将PE文件在内存中的实际映像地址减去PE文件所要求的映像地址，得到一个差值，再将这一差值根据重定位类型的不同添加到地址数据中。

10.9　资源

Windows程序的各种界面称为资源，包括加速键（Accelerator）、位图（Bitmap）、光标（Cursor）、对话框（Dialog Box）、图标（Icon）、菜单（Menu）、串表（String Table）、工具栏（Toolbar）、版本信息（Version Information）等。在PE文件所有结构中，资源部分是最复杂的。

10.9.1　资源结构

资源用类似于磁盘目录结构的方式保存，目录通常包含3层。最上面的目录很类似于一个文件系统的根目录。每一个在根部下的目录条目总是在它自己权利下的一个目录。这些二级目录中的每一个对应于一个资源类型（字符串表、菜单、对话框、菜单等）。在每一个二级资源类型目录下，是第三级子目录。目录结构如图10.26所示。

[image]

图10.26　资源的树形结构

1．资源目录结构

数据目录表中的IMAGE_DIRECTORY_ENTRY_RESOURCE条目包含资源的RVA和大小。资源目录结构中的每一个节点都是由IMAGE_RESOURCE_DIRECTORY结构和紧跟其后的数个IMAGE_RESOURCE_DIRECTORY_ENTRY结构组成的，这两种结构组成了一个目录块。

IMAGE_RESOURCE_DIRECTORY结构长度为16字节，共有6个字

段。其定义如下所示：

[image]

在这个结构中唯一令人感兴趣的字段是NumberOfNamedEntries和NumberOfIdEntries，它们说明了本目录中目录项的数量。NumberOfNamedEntries字段是以字符串命名的资源数量，NumberOfIdEntries是以整型数字来命名的资源数量，两者加起来是本目录中的目录项总和，即为后面紧跟的IMAGE_RESOURCE_DIRECTORY_ENTRY数目。

2．资源目录入口结构

紧跟着资源目录结构后的就是资源目录入口（Resource Dir Entries）结构，此结构长度为8个字节，包含2个字段。IMAGE_RESOURCE_DIRECTORY_ENTRY结构定义如下：

[image]

根据不同情况，这两个字段的含义不一样。

（1）Name字段

该字段定义目录项的名称或ID。当结构用于第一层目录时，定义的是资源类型；当结构用于第二层目录时，定义的是资源的名称；当结构用于第三层目录时，定义的是代码页编号。当最高位是0时，表示字段的值作为ID使用；而最高位为1时，字段的低位作为指针使用，资源名称字符串是使用UNICODE编码，这个指针并不直接指向字符串，而是指向一个IMAGE_RESOURCE_DIR_STRING_U结构。其定义如下：

[image]

（2）OffsetToData字段

该字段是一个指针。当最高位（位31）为1时，低位数据指向下一层目录块的起始地址；当最高位为0时，指针指向IMAGE_RESOURCE_DATA_ENTRY结构。

当将Name和OffsetToData用做指针时需要注意，该指针是从资源区块开始的地方算起的偏移量，不是RVA，即根目录的起始位置的偏移量。

有一点要说明的是，当IMAGE_RESOURCE_DIRECTORY_ENTRY用在第一层时，它的Name字段作为资源类型使用。当资源类似以ID定义并且数值在1到16之间时，表示是系统预定义的类型，具体见表10-16。

表**10-16**　系统预定义资源类型

[image]

3．资源数据入口

经过三层IMAGE_RESOURCE_DIRECTORY_ENTRY（一般是3层，也

有可能更少些。第一层是资源类型，第二层是资源名，第三层是资源的Language），第三层目录结构中的OffsetToData指向IMAGE_RESOURCE_DATA_ENTRY结构。该结构描述了资源数据的位置和大小，其结构定义如下：

[image]

经过多层结构后，此处的IMAGE_RESOURCE_DATA_ENTRY结构就是真正的资源数据了。结构中的OffsetToData指向资源数据的指针，其为RVA值。

10.9.2　资源结构实例分析

在这里以光盘映像文件中提供的实例pediy.exe分析资源。数据目录表的第三个成员指向资源结构，该指针具体位置是在PE文件头的88h偏移处。用十六进制工具查看实例文件的PE文件头起始位置是0C0h，则资源结构在整个文件的0C0h + 88h = 148h处，因此在148h处可以发现资源的RVA为4000h。由于这个实例文件磁盘文件中的区块对齐值等于1000h，与内存页对齐值相同，因此RVA与文件偏移地址不用转换。图10.27所示就是该程序资源的一部分，文件偏移4000h是资源起始地址。

[image]

图10.27　资源的十六进制形式

1．根目录

文件偏移4000h处指向的是根目录，第一行数据就是IMAGE_RESOURCE_DIRECTORY结构，其各项的值如图10.28所示。

[image]

图10.28　根目录的IMAGE_RESOURCE_DIRECTORY结构

从图10.28读出根目录的IMAGE_RESOURCE_DIRECTORY各结构成员值：Characteristics为00000000，TimeDateStamp为00000000，MajorVersion为0000，MinorVersion为0000，NumberOfNamedEntries为0000，NumberOfIdEntries为0003。NumberOfNamedEntries与NumberOfIdEntries的和是3，表明这个程序有3个资源项目。也就是说，其后面紧跟着3个IMAGE_RESOURCE_DIRECTORY_ENTRY结构，根据图10.27中的数据将这3个结构整理见表10-17。

表**10-17**　根目录下的**IMAGE_RESOURCE_DIRECTORY_ENTRY**结构数据

[image]

以表10-17中的第二个IMAGE_RESOURCE_DIRECTORY_ENTRY结

构为例分析资源的下一层。第一层目录，Name字段是定义资源类型，目前其ID值为04h，表明这是一个菜单资源。另外，OffsetToData字段为80000040h，第一个字节80h的二进制为10000000，最高位为1，说明还有下一层。所以OffsetToData的低位数据40h指向第二层目录块。第二层目录块的地址为资源块首地址加上40h，即为4000h + 40h = 4040h。

2．第二层目录

偏移4040h的数据即为第二层，见图10.29。

[image]

图10.29　第二层目录的IMAGE_RESOURCE_DIRECTORY结构

图10.29中阴影部分是第二层的IMAGE_RESOURCE_DIRECTORY结构成员：Characteristics为0，TimeDateStamp为0，MajorVersion为0，MinorVersion为0，NumberOfNamedEntries为1，NumberOfIdEntries为0。NumberOfNamedEntries与NumberOfIdEntries的和是1，表明这层有一个资源项目。也就是说，其后面紧跟着1个IMAGE_RESOURCE_DIRECTORY_ENTRY结构，即在文件偏移4050h处，Name是800000E8h，OffsetToData是80000088h。

当在第二层目录时，Name字段定义的是资源名称，Name字段第一个字节80h的二进制为10000000，最高位为1，表明这是一个指针，指向IMAGE_RESOURCE_DIR_STRING_U结构，其地址为资源块首地址加上Name字段低位数据0E8h，即4000h + 0E8h = 40E8h。具体见图10.30中阴影部分，图中显示Length是05，NameString是Unicode字符“PEDIY”，即这个资源名为“PEDIY”。

[image]

图10.30　IMAGE_RESOURCE_DIR_STRING_U结构

OffsetToData字段是80000088h，第一个字节80h的二进制为10000000，最高位为1，说明还有下一层。所以OffsetToData的低位数据88h指向第三层目录块。第三层目录块的地址为资源块首地址加上88h，即为4000h + 88h = 4088h。

3．第三层目录

文件偏移4088h的数据指向第三层，如图10.31所示。

[image]

图10.31　第三层目录的IMAGE_RESOURCE_DIRECTORY结构

从图10.31中可以看到第三层的IMAGE_RESOURCE_DIRECTORY结构成员：Characteristics为0，TimeDateStamp为0，MajorVersion为0，MinorVersion为0，NumberOfNamedEntries为0，NumberOfIdEntries

为1。NumberOfNamedEntries与NumberOfIdEntries的和是1，表明这层有一个资源项目。也就是说，其后面紧跟着1个IMAGE_RESOURCE_DIRECTORY_ENTRY结构，即在文件偏移4098h处，Name是00000409h，OffsetToData是000000C8h。

当在第三层目录时，Name字段定义的是代码页编号，00000409h表示代码页是英语。OffsetToData高位现在是0，所以其低位数据0C8h指向IMAGE_RESOURCE_DATA_ENTRY结构，0C8h加上资源块首地址，即4000h + 0C8h = 40C8h，见图10.32中阴影部分。

[image]

图10.32　IMAGE_RESOURCE_DATA_ENTRY结构

在这里就能查看到IMAGE_RESOURCE_DATA_ENTRY结构成员值，OffsetToData是00004400，Size是0000005Ah，CodePage是0000000，Reserved是00000000h。此时图标的真正资源数据RVA为4400h，大小为5Ah。

10.9.3　资源编辑工具

资源数据，它们一般被存储在PE文件的.rsrc区块中，并且不能通过由程序源代码定义的变量直接访问，Windows提供函数直接或间接地把它们加载到内存中以备使用。

资源类型主要有以下几种：

- VC类标准资源（包括菜单、对话框、串表等资源）；
- Delphi类标准资源（Rcdata资源）；
- 非标准的Unicode字符（主要是一些VB编译程序等）。

这些资源可以定制和修改，如更改字体、对话框，增加按钮、菜单等。Visual C＋＋等编译器可以直接编辑修改PE文件的资源。另外常用的资源修改工具有Resource Hacker和eXeScope等，它们也可直接编辑修改用VC＋＋及Delphi编制的程序资源，包括EXE、DLL和OCX等，并且是功能强大的汉化和调试辅助工具。

10.10　TLS初始化

使用线程本地存储器（TLS）可以将数据与执行的特定线程联系起来。当使用__declspec(thread)声明的TLS变量时，编译器将它们放入一个叫.tls的区块里。当应用程序加载到内存中时，系统要寻找可执行文件中的.tls区块，并且动态地分配一个足够大的内存块，以便存放所有的TLS变量。系统也将一个指向已分配的内存的指针放到TLS数组里，这个数组由FS:[2Ch]指向（在x86架构上）。

在一个可执行文件中的线程局部存储（TLS）数据是由数据目录表中的IMAGE_DIRECTORY_ENTRY_TLS条目指出的。如果是非零的，这个字段指向一个IMAGE_TLS_DIRECTORY结构。其结构如下：

[image]

AddressOfCallBacks是线程建立和退出时的回调函数，包括主线程和其他线程。当一个线程被创建或销毁时，在列表中的每一个函数被调用。一般程序都没有回调函数，这个列表是空的。有一点特别注意，程序运行时，TLS数据初始化和TLS回调函数都在入口点（AddressOfEntryPoint）之前执行，也就是说，TLS是程序最开始运行的地方，许多病毒或外壳程序就利用这点执行一些特殊操作。程序退出时，TLS回调函数再被执行一次。

在IMAGE_TLS_DIRECTORY结构中的地址是虚拟地址，而不是RVA。这样，如果可执行文件不是从基地址装入，那么这些地址就会通过基址重定位来修正，而且，IMAGE_TLS_DIRECTORY本身并不在.tls区块中，它存在于.rdata区块里。

10.11　调试目录

当用调试信息构建一个可执行文件时，按照惯例应该包括这种信息的格式及位置细节。操作系统并不需要这个来运行可执行文件，但这对开发工具是有用的。

数据目录表的第7个条目（IMAGE_DIRECTORY_ENTRY_DEBUG）指向调试目录，它由一个IMAGE_DEBUG_DIRECTORY结构数组组成。这些结构保持存储在文件中的变量的类型、尺寸和位置的调试信息。debug目录里的元素数量可以通过DataDirectory内的Size字段来计算。其结构定义如下：

[image]

到目前为止，debug信息的最普遍形式是PDB文件。PDB文件基本上是CodeView样式的debug信息的演变。PDB信息的存在是由一个IMAGE_DEBUG_TYPE_CODEVIEW类型的调试目录字段指出的。如果检查由这个条目指向的数据，将发现一个简短的CodeView样式的头部。这个debug数据的多半是指向外部PDB文件的路径。在Visual Studio 6.0中，debug头部以NB10标识开始，在Visual Studio .NET中，头部是以RSDS开始的。

在Visual Studio 6.0中，COFF格式的debug信息能用/DEBUGTYPE:COFF链接器开关生成。在Visual Studio .NET中这个选项没有了。

10.12　延迟装入数据

延迟装入一个DLL是一种混合方式，其是通过LoadLibrary和GetProcAddress获得延迟加载函数的地址，然后直接转向对延迟加载函数的调用。

记住延迟装入不是操作系统的一个特征，它完全是通过链接器和运行库加入额外的代码和数据来实现的。同样地，无法在winnt.h里找到关于延迟装入的更多参考，不过，可以在延迟装入数据和常规的输入数据之间看到一定的相似之处。

数据目录表中的IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT条目

指向延迟装入的数据，这是一个指向ImgDelayDescr结构数组的RVA，这个结构定义在Visual C＋＋的DelayImp.H中，表10-18说明了它的内容。每一个被延迟装入的DLL都对应一个ImgDelayDescr结构。

表10-18　ImgDelayDescr结构

[image]

ImgDelayDescr结构的关键在于它包括了对应DLL的IAT和INT的地址，这些表在格式上与常规的输入表是相同的，唯一的区别是它们由运行库代码而不是操作系统进行写入和读出的。当第一次从一个延迟装入的DLL中调用一个API函数时，运行库调用LoadLibrary（如果需要），然后是GetProcAddress，最后得到的地址被存在延迟装入的IAT表中，这样以后每次调用这个API都会直接到这里来。

在Visual C＋＋ 6.0中有延迟装入数据最初的原型，ImgDelayDescr中所有包含地址的域均是虚拟地址，而不是RVA。换句话说，它们包含延迟装入数据所在位置的实际地址。这些域是双字，是x86上一个指针的大小。现在向IA-64的快速移植正在被支持。很显然4字节不足以装下一个完整的地址，在这点上，微软做了件正确的事情，已将包含地址的字段改为了RVA。表10-18中已经用了修正的结构定义和名称。

关于在ImgDelayDescr结构中是使用RVA还是虚拟地址，现在仍有争论，在这个结构中有一个字段是旗标值。当grAttrs字段被设为1时，结构成员被当成是RVA。这是唯一与Visual Studio .NET和64位编译器一起出现的选项。如果在grAttrs中的这个位关掉，ImgDelayDescr字段将是虚拟地址。

10.13　程序异常数据

一些体系结构（包括IA-64）不使用基于框架的异常处理，像在x86上就是这样。取而代之的是，它们使用基于表的异常处理。这种方式下，里面有一个表，它包含了每一个有可能受异常展开影响的函数信息。为每个函数准备的数据包括起始地址、结束地址以及关于异常应该如何处理并在什么地方被处理的信息。当一个异常发生时，系统通过遍历这个表来定位合适的入口并处理它。异常表是一个IMAGE_RUNTIME_FUNCTION_ENTRY结构数组，数组是由数据目录表中的IMAGE_DIRECTORY_ENTRY_EXCEPTION条目来指向的。IMAGE_RUNTIME_FUNCTION_ENTRY结构的格式随体系结构的不同而不同。对于IA-64，布局看上去像这个样子：

[image]

UnwindInfoAddress数据的格式没有在winnt.h中给出，但是，可以从Intel的“IA-64 Software Conventions and Runtime Architecture Guide”这份资料的第11章中找到。

10.14　.Net头部

.Net文件为Microsoft .Net环境生成的可执行文件。.Net环境由公共语言运行环境（CLR）和.Net框架类库组成。可以把CLR看作是一台虚拟机，.Net应用程序就在这台机器中运行。.Net可执行文件的主要目的是获得.Net特定的装入内存的信息，像元数据（Metadata）和中间语言（Intermediate Language，简称IL）。另外，.Net可执行文件依靠MSCOREE.DLL进行链接，这个DLL对于一个.Net进程是起始点。当一个.Net可执行文件装入时，它的入口点通常是一小块残余代码，这小块代码只不过是跳到MSCOREE.DLL中的一个输出函数（_CorExeMain或_CorDllMain）。从那里，MSCOREE接管并开始使用来自可执行文件的元数据和中间语言。这种运行类似于Visual Basic程序使用MSVBVM60.DLL的方式。

.Net环境下的PE文件，在整体结构上与传统PE一致，所不同的，就是利用了数据目录表中的IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR条目扩充了其结构。这个条目原本是设计用于COM，但一直没有被使用，现在用于保存.Net的信息结构，指向IMAGE_COR20_HEADER。有关.Net的具体内容，请参考第9章。

10.15　编写PE分析工具

常用的PE分析工具有LordPE、Stud_PE等，熟悉PE格式后，这些工具使用都比较简单，读者可自行摸索。本节将从学习角度，讲解如何编写一个简单的PE结构分析程序，这对理解PE格式和相关PE编程非常有帮助。

PE格式分析工具的编写并不难，主要就是对PE格式的各个结构进行定位。本节定义了一个MAP_FILE_STRUCT结构来存放有关信息。结构如下：

[image]

10.15.1　文件格式检查

文件格式可以通过PE header开始的标志Signature来检测。也许读者会说，检测DOS Header的Magic Mark不是也可以检测此PE文件是否合法吗？这个想法没有错，但是检测Magic Mark不一定能确定就是PE文件，如果某文本文件正好在开始就是“MZ”字符串，就会误判断。

（1）判断文件开始的第一个字段是否为IMAGE_DOS_SIGNATURE，即5A4Dh。

（2）再通过e_lfanew找到IMAGE_NT_HEADERS，判断Signature字段的值是否为IMAGE_NT_SIGNATURE，即00004550h，如果是IMAGE_NT_SIGNATURE，就可以认为该文件是PE格式。

具体实现的代码如下：

[image]

10.15.2　FileHeader和OptionalHeader内容的读取

只要得到了IMAGE_NT_HEADERS，根据IMAGE_NT_HEADERS的定义，就可以找到IMAGE_FILE_HEADER和PIMAGE_OPTIONAL_HEADER。

（1）得到IMAGE_NT_HEADERS结构指针的函数：

[image]

（2）得到IMAGE_FILE_HEADER结构指针的函数：

[image]

（3）得到IMAGE_OPTIONAL_HEADER结构指针的函数：

[image]

得到指向IMAGE_NT_HEADERS结构和IMAGE_OPTIONAL_HEADER结构指针的函数以后，接下来就是要把FileHeader和OptionalHeader的信息显示出来。例如，要把FileHeader和OptionalHeader的信息以十六进值方式显示在编辑控件上，此时先用函数wsprintf()将欲显示的值进行格式化，然后再调用API函数SetDlgItemText即可。具体代码如下：

[image]

10.15.3　得到数据目录表信息

数据目录表（DataDirectory）由一组数组构成，每组项目包括执行文件的重要部分的起始RVA和长度。因为数据目录有16项，如果不嫌麻烦，代码可以一行行的写，在这里定义了一个编辑控件ID的结构数组，用一个循环就可以了。具体代码如下：

[image]

10.15.4　得到区块表信息

紧接IMAGE_NT_HEADERS以后就是区块表（Section Table）了，Section Table则是由IMAGE_SECTION_HEADER组成的数组。如何得到Section Table的位置呢？换句话说，也就是如何得到第一个IMAGE_SECTION_HEADER的位置。在Visual C++中，可以利用IMAGE_FIRST_SECTION宏来轻松地得到第一个IMAGE_SECTION_HEADER的位置。

又因为区块的个数已经在文件头中指明了，所以只要得到第一个区块的位置，然后再利用一个循环语句就可以得到所有区块的信息了。

下面的GetFirstSectionHeader函数是利用IMAGE_FIRST_SECTION宏得到区块表的起始位置。

[image]

这里必须要强调一下，在一个PE文件中，OptionalHeader的大小是

可以变化的，虽然它的大小通常为E0h，但是总有例外。原因是可选文件头的大小是由文件头中的SizeOfOptionalHeader字段指定的，并不是个固定值。这也是IMAGE_FIRST_SECTION宏对于可选文件头的大小为什么不直接用固定值的原因。系统的PE加载器在加载PE文件的时候，也是利用了文件头中的SizeOfOptionalHeader字段的值来定位区块表的，而不是用固定值。能否正确地定位到区块表，取决于SizeOfOptionalHeader字段的值的正确性。这是个很容易被忽略的问题，因此会导致一些程序的BUG，如Peditor v1.7和PEiD v0.9都有这样的BUG。本例中，用ListView控件来显示PE文件中的区段信息。具体代码如下：

[image]

10.15.5　得到输出表信息

输出表（Export Table）中的主要成分是一个表格，内含函数名称、输出序数等。输出表是数据目录表的第一个成员，其指向IMAGE_EXPORT_DIRECTORY结构。输出函数的个数由结构IMAGE_EXPORT_DIRECTORY的字段NumberOfFunctions来说明。实际上，也有例外。例如，在写一个DLL的时候，可以用DEF文件来制定输出函数的名称、序号等。请看下面这个DEF文件的内容：

[image]

在这个文件中，共输出了11个函数（func1 ~ func11），而输出函数的序号却是从1至31，如果没有考虑这一点的话，很有可能会在这里出错。因为这时IMAGE_EXPORT_DIRECTORY的字段NumberOfFunctions值为0x1F，即31。如果认为NumberOfFunctions值就为输出函数个数的话，那就错了。图10.33所示的就是程序出错时的界面。

[image]

图10.33　显示输出函数信息

在这里使用十六进制工具来分析一下这个DLL，如图10.34所示。11个小框中的数据才是真正的输出函数的RVA，其余的所谓的“输出函数的RVA”则用0填充了。

[image]

图10.34　用十六进制工具分析

因此编程时，必须将这些特殊情况考虑进去，正确显示输出表和输出函数信息的程序代码见光盘映像文件中的ShowExportFuncsInfo(HWND hDlg)。

10.15.6　得到输入表信息

数据目录表第二成员指向输入表。输入表以一个IMAGE_IMPORT_DESCRIPTOR结构开始，以一个空的IMAGE_IMPORT_DESCRIPTOR结构结束。在这里可以通过GetFirstImportDesc函数得到Import Table在文件中的位置。GetFirstImportDesc函数的定义如下：

[image]

找到了输入表的位置，可以通过一个循环来得到整个输入表，循环终止的条件是IMAGE_IMPORT_DESCRIPTOR结构为空。请看ShowImportDescInfo函数的定义：

[image]

[image]

在ShowImportDescInfo函数中，首先用GetFirstImportDesc函数得到指向第一个IMAGE_IMPORT_DESCRIPTOR结构的指针pImportDesc，以pImportDesc－＞FirstThunk为真来作为循环的条件，循环得到Import Table的各项信息。

通过上面的ShowImportDescInfo函数，可以得到PE文件所引入的DLL的信息，接下来的任务就是如何分析得到通过DLL所输入的函数的信息，这里必须通过IMAGE_IMPORT_DESCRIPTOR所提供的信息来得到输入的函数信息。具体代码见光盘映像文件中的ShowImportFuncsByDllIndex(HWND hDlg,int index)函数。可以通过名字和序号来引入所用的函数，怎么来区分一个函数是如何引入的呢？在于IMAGE_THUNK_DATA值的高位，如果被置位了，低31位被看作是一个序数值。如果高位没被置位，IMAGE_THUNK_DATA值是一个指向IMAGE_IMPORT_BY_NAME的RVA。如果两者都不是，则可以认为IMAGE_THUNK_DATA值为函数的内存地址。

另外，微软的ImageHlp库中提供了大量的有关对PE Image操作的API，直接调用就可，当然这些API的功能读者也可以自己来实现。

第11章　结构化异常处理

结构化异常处理（Structured Exception Handling，简称SEH）是Windows操作系统处理程序错误或异常的技术。SEH是Windows操作系统的一种系统机制，与特定的程序设计语言无关。本章仅简单地从调试角度认识一下SEH，有关SEH更深层的知识，请参考《软件加密技术内幕》一书中温玉杰撰写的“第4章Windows下的异常处理”或其他相关资料。

11.1　基本概念

Intel公司在从386开始的IA-32家族处理器中引入了中断（Interrupt）和异常（Exception）的概念。中断是由外部硬件设备或异步事件产生的。异常是由内部事件产生的，异常又可分为故障、陷阱和终止三类。故障和陷阱，正如其名称所暗示的，是可恢复的；终止类异常是不可恢复的，如果发生了这种异常，系统必须重启。

11.1.1　异常列表

所谓异常就是在应用程序的正常执行过程中发生的不正常事件。CPU引发的异常称为硬件异常，例如访问一个无效的内存地址。操作系统或应用程序引发的异常称为软件异常。表11-1是常见异常的列表。

表11-1　异常列表

[image]

除了CPU捕获一个事件并引发一个硬件异常外，在代码中也可以强制引发一个软件异常。只需调用RaiseException函数：

[image]

程序捕获软件异常的方法与捕获硬件异常的完全相同。

11.1.2　异常处理的基本过程

首先来看看一个应用程序发生错误后，Windows是如何结合SEH机制进行处理的。

（1）因为有多种异常，系统首先判断异常是否应发送给目标程序，如果应该发送，并且目标程序正处于被调试状态，则系统挂起程序，填写如下结构：

[image]

将成员dwFirstchance置为1，并向调试器发送EXCEPTION_DEBUG_EVENT消息。剩下的事情就由调试器全权负责了，调试器可能处理这个异常，也可能无法处理。

（2）如果调试器未能处理异常或程序根本没有被调试，系统就会查找是否存在与线程相关的异常处理过程，如果目标程序中存在与线程相关的异常处理过程，系统就调用程序的线程相关的SEH异常处理例程，交由其处理。

（3）与线程相关的异常处理过程可以有一个或多个，每个可以选择处理或者不处理异常，如果它不处理并且存在多个线程相关的异常处理过程，可交由链起来的其他异常处理过程进行处理，依此类推。

（4）如果程序线程的异常处理均选择不处理异常，如果程序处于被调试状态，操作系统仍会再次挂起程序通知调试器，这次EXCEPTION_DEBUG_INFO结构的dwFirstchance成员置为0。

（5）如果程序未处于被调试状态或者调试器仍然未能够处理，并且程序调用了API函数SetUnhandledExceptionFilter设置了与进程相关的异常处理过程的话，系统转向对它的调用。

（6）如果程序没有设置进程相关的异常处理过程或者进程相关的异常处理过程也未能处理这个异常，系统会调用默认的系统异常处理程序，通常显示一个对话框（如图11.1所示），可以选择“确定”或者最后将其附加到调试器上的“取消”按钮。如果没有调试器能被附加于其上或调试器还是处理不了异常，系统就调用ExitProcess终结程序。

[image]

图11.1　Windows XP下的出错对话框

（7）不过在终结之前，系统再次调用发生异常的线程中所有的异常处理过程，这是线程异常处理过程获得的最后清理未释放资源的机会，其后程序就终结了。

学习SEH要树立一个最基本的观念：SEH是系统发现异常或错误时，在终结应用程序之前给应用程序的一个最后改正错误的机会，从程序设计的角度来说，就是系统在终结程序之前给程序的一个执行其预设定的回调函数的机会。

11.1.3　SEH的分类

一般地，按作用域（即其监视范围）分，SEH可分为两类：一类是监视某线程中某段代码是否发生异常的异常处理过程，一般称为线程相关的异常处理过程，或称为Per_Thread类型的异常处理过程，有时也简称为线程异常处理。另一类是监视整个进程中所有线程是否发生异常的异常处理过程，称为进程相关的异常处理过程，或称为“Final”型异常处理过程。有人也称之为筛选器，源于其对应于设置其的API函数SetUnhandledExceptionFilter中的Filter一词，在Win32 API文档中，称之为顶层（top-level）异常处理。

11.2　SEH相关数据结构

SEH涉及了几个关键的数据结构。

11.2.1　TEB结构

TEB（Thread Environment Block，线程环境块）在Windows 9x系列中称为TIB（Thread Information Block，线程信息块），它记录着线程的重要信息。每一个线程对应一个TEB结构，在Windows 2000 DDK中定义为：

[image]

与异常处理相关的项是指向EXCEPTION_REGISTRATION结构的指针ExceptionList，正好位于TEB的偏移0处。Windows在创建线程时，操作系统均会为每个线程分配TEB，而且都将FS段选择器指向当前线程的TEB数据（单CPU机器上的系统中在任何时刻只有一个线程在执行），这就为程序提供了存取TEB数据的途径，即TEB总是由[FS:0]指向的。

11.2.2　EXCEPTION_REGISTRATION结构

TEB偏移量为00h的_EXCEPTION_REGISTRATION_RECORD主要用于描述线程异常处理过程的地址，多个该结构的链表描述多个线程异常处理过程的嵌套层次关系。其定义如下：

[image]

其中，prev是指向前一个EXCEPTION_REGISTRATION（简称ERR）的指针，形成一链状结构；handler指向异常处理代码，如图11.2所示。当系统遇到一个它不知如何处理的异常时，它就查找异常处理链表。每个线程都有它自己的异常处理链表。

[image]

图11.2　异常处理链表的示意图

本书实例seh.exe程序演示了一个结构异常处理的例子，用OllyDbg打开，注意OllyDbg的“调试选项/异常”设置，所有选项不勾选，按F9键让seh.exe跑起来，程序将在401017这行停住。

[image]

如果程序试图读取或写入0地址空间，那么CPU就会引发一个访问违规。在Win98下，这个地址空间是0～0FFFh，在NT架构下是0～FFFFh，这个地址空间是禁止进入的，如果读取就会发生内存访问违规现象。由于访问违规，OllyDbg状态栏就有如一行字符提示，“访问违规：读取[00000000] -使用Shift＋F7/F8/F9忽略程序异常”，如图11.3所示。按Shift＋F9键即可忽略异常继续执行。

[image]

图11.3　OllyDbg状态栏里提示是否忽略异常

如果在OllyDbg的“调试选项/异常”设置忽略“非法访问内存”异常，

OllyDbg遇到此类异常将不暂停。OllyDbg这种对异常处理的设置，非常有利于程序的跟踪，平时要擅于使用。

接下来，从调试器的角度来理解一下SEH的机制。在OllyDbg选项里忽略所有异常，重新加载实例seh.exe调试。相关汇编代码如下：

[image]

[image]

跟踪SEH程序一定要注意堆栈窗口，以查看handler的值，然后对此地址设断。用OllyDbg跟踪到00401017地址处时，查看堆栈窗口的数据，如图11.4所示。

[image]

图11.4　查看ERR结构

此时，handler值为40102E，对40102E设断，然后按F9键让程序运行。当执行00401017一句读取线性地址0时，产生异常，将跳到40102E（handler）处继续执行。因此，只有提前在handler地址处设置断点，才能正常跟踪SEH代码的运行，否则代码就有可能跟“飞”。

11.2.3　EXCEPTION_POINTERS、EXCEPTION RECORD、CONTEXT

当一个异常发生时，操作系统向引起异常的线程的堆栈里压入EXCEPTION_POINTERS结构，EXCEPTION_POINTERS结构包含两个指针，一个指向EXCEPTION_RECORD结构，一个指向CONTEXT结构。

[image]

上例跟踪到00401017地址处，按F7键或Shift + F7键就能进入异常刚发生时的系统代码处，EXCEPTION_POINTERS结构就在栈顶，即ESP -> ptEXCEPTION_POINTERS。当前堆栈的数据就是EXCEPTION_POINTERS结构的两个数据成员EXCEPTION_RECORD和CONTEXT，如图11.5所示。

[image]

图11.5　EXCEPTION_POINTERS结构

1．EXCEPTION_RECORD结构

EXCEPTION_RECORD结构包含有关最近发生异常的详细信息，这些信息独立于CPU。其结构如下：

[image]

其中ExceptionCode字段定义了产生异常的原因，表11-2列出了一

些常见的异常原因。当然也可以定义自己的ExceptionCode。

表**11-2**　常见的异常原因代码列表

[image]

上例中的“mov eax, [esi]”指令读取0地址，所以会引发一个EXCEPTION_ACCESS_VIOLATION异常，异常代码是0C0000005h。

在图11.5所示处，查看OllyDbg数据窗口12FCD4处的数据，即获得EXCEPTION_RECORD结构的数据成员（见图11.6），显示异常代码是0C0000005h，异常发生地址是401017h。为了直观显示，也可在OllyDbg数据窗口执行右键菜单中的“long/Hex”（长型/十六进制）将数据切换成DWORD显示。

[image]

图11.6　EXCEPTION_RECORD结构

2．CONTEXT结构

CONTEXT结构是Win32 API一个几乎唯一与处理器相关的结构，包括了线程运行时处理器各主要寄存器的完整镜像，用于保存线程运行时环境。目前，已经存在为Intel、MIPS、Alpha和PowerPC处理器定义的CONTEXT结构。若要了解这些结构，可参考VC的头文件WinNT.h。

x86 CPU的CONTEXT结构如下：

[image]

该结构的大部分域是不言自明的，值得解释的是其第一个域ContextFlags，它表示该结构中的哪些域有效，当需要在用CONTEXT结构保存的信息恢复执行时可对应更新，这为有选择地更新部分而非全部提供了手段。

以下是摘自Windows.inc中的定义：

[image]

上述结构就是在处理SEH时用到的大部分结构，通常作为参数传递给异常处理回调函数，可以通过这些结构了解相关信息或改变线程的状态。

在图11.5所示处，查看OllyDbg数据窗口12FCF0处的数据，即获得CONTEXT结构的数据成员（见图11.7），该结构储存的是图11.4所示刚触发异常时各种寄存器信息。

[image]

图11.7　CONTEXT结构

例如，“D 12FCF0 + 0B8”显示刚发生异常时的地址（EIP）为

401017。

CONTEXT结构非常重要，程序通过修改CONTEXT结构中的成员，可以干出许多“出格”的事情，以达到反跟踪的目的。

11.3 异常处理回调函数

SEH异常处理回调函数的参数定义如下：

[image]

返回值：

ExceptionContinueExection（等于0）：回调函数返回后，系统将线程环境设置为_lpContext参数指定的CONTEXT结构并继续执行；

ExceptionContinueSearch（等于1）：回调函数拒绝处理这个异常，系统将通过EXCEPTION_REGISTRATION结构的prev字段得到前一个回调函数的地址并调用它；

ExceptionNestedException（等于2）：回调函数在执行中又发生了新的异常，即嵌套异常；

ExceptionCollidedUnwind（等于3）：发生了嵌套的展开操作。

CONTEXT结构成员可以改变，这意味着程序可以用改变的CONTEXT内容去执行程序，如清除断点，改变代码运行路线等。

例如，本书实例中seh2.exe实例的部分代码如下：

[image]

[image]

该实例通过修改CONTEXT结构中的成员，将调试寄存器Drx清零，使断点失效，以达到反跟踪的目的。

跟踪时可按上一节的方法从ERR结构入手，操作过程如下。

① 执行到00401018一行，记下堆栈中的handler值00401051。

② 按F7键进入系统代码，查看CONTEXT.EIP的值，命令行为“D [esp + 4] + 0B8”，因为软件保护中一般会通过CONTEXT.EIP的值改变程序流程。

③ 一步步执行，直到CONTEXT.EIP值改变。40105E这行命令“mov dword ptr [eax + 0B8], ebx”就是对CONTEXT.EIP赋值。

④ 程序处理完后，将跳到新的CONTEXT.EIP（0040102D）代码处执行。

⑤ 程序在处理CONTEXT结构的同时，将Dr0，Dr1，Dr2和Dr3调试寄存器清零，使调试器的断点失效等，以达到反跟踪的目的。

跟踪时的具体汇编代码如下：

[image]

[image]

第6篇　脱壳篇

- 第12章　专用加密软件
- 第13章　脱壳技术

现在越来越多的软件都采用了加壳保护。在软件分析和汉化过程中，脱壳是必不可少的一步，本章详细介绍了基本的脱壳技术，完全为脱壳新手量身定做。

第12章　专用加密软件

软件加密的资料相对来说比较匮乏，它是一种对抗性的技术，需要开发者对解密技术有一定了解。对于大多数开发者来说，由于不熟悉加密与解密这个领域，导致花费了大量人力和物力设计的保护方案不堪一击。术业有专攻，为了让软件开发者从软件保护措施中脱离出来，专心致力于自己的软件开发，出现了一个新的事物——专用加密软件。

12.1　认识壳

壳①是最早出现的一种专用加密软件技术，现在越来越多的软件都加壳保护，那到底什么是壳呢？

12.1.1　壳的概念

在自然界中，植物用壳来保护种子，动物用壳来保护身体等。同样，在一些计算机软件里也有一段专门负责保护软件不被非法修改或反编译的程序。它们附加在原程序上通过Windows加载器载入内存后，先于原始程序执行，得到控制权，执行过程中对原始程序进行解密、还原，还原完成后再把控制权交还给原始程序，执行原来的代码的部分。加上外壳后，原始程序代码在磁盘文件中一般是以加密后的形式存在的，只在执行时在内存中还原，这样就可以比较有效地防止破解者对程序文件的非法修改，同时也可防止程序被静态反编译。由于这段程序和自然界的壳在功能上有很多相同的地方，基于命名的规则，就把这样的程序称为“壳”了，如图12.1所示。

[image]

图12.1　描述壳的示意图

最早提出“壳”这个概念的，是当年推出脱壳软件RCOPY 3的作者熊焰先生。在DOS时代，“壳”一般都是指磁盘加密软件的段加密程序，可能是那时候的加密软件还刚起步不久，所以大多数的加密软件（加壳软件）所生成的“成品”在“壳”和需要加密的程序之间总有一条比较明显的“分界线”。有经验的人可以在跟踪软件的运行以后找出这条分界线来。

脱壳技术的进步，促进且推动了当时的加壳技术的发展。LOCK95和BITLOK等所谓的“壳中带籽”加密程序纷纷出笼，真是各出奇谋，把小小的软盘也折腾得够辛苦的了。国内的加壳软件和脱壳软件正在较量得火红的时候，国外的“壳”类软件早已经发展到像LZEXE之类的压缩壳了。这类软件其实就是一个标准的加壳软件，它把EXE文件压缩了以后，再在文件上加上一层在软件执行时自动把文件解压缩的“壳”来达到压缩EXE文件的目的。接着，这类软件越来越多，比如PKEXE、AINEXE、UCEXE和WWPACK都属于这类软件。奇怪的是，当时看不到一个国产的同类软件。

过了一段时间，可能是国外淘汰了磁盘加密，转向使用软件序列号加密方法，保护EXE文件不被动态跟踪和静态反编译就显得非常重要了。所以专门实现这样功能的加壳程序便诞生了。MESS、CRACKSTOP、HACKSTOP、TRAP、UPS等都是比较有名气的本类软件代表。这样的软件才能算是正宗的加壳软件。

由于Microsoft保留了Windows 95的很多技术上的秘密，即便Windows 95已经推出三年多的时间，也没见过在其上面运行的“壳”类软件。直到1998年的中期，这样的软件才迟迟出现，而这个时候Windows 98也发表了一段日子。这类的软件不发表尚可，一发表就大批地涌现出来。先是加壳类的软件，如BJFNT、PELOCKNT等，它们的出现，使暴露三年多的Windows下的PE格式EXE文件得到很好的保护。接着出现的就是压缩壳（Packers），因为Windows下运行的EXE文件“体积”一般都比较大，所以它的实用价值比起DOS下的压缩软件要大很多。这类软件也很多，UPX、ASPack、PECompact等都是其中的佼佼者。随着软件保护的需要，出现了加密壳（Protectors），它用上了各种反跟踪技术保护程序不被调试、脱壳等，其加壳后的体积大小不是其考虑的主要因素，如ASProtect、Armadillo、EXECryptor等。随着加壳技术的发展，这两类软件之间的界线越来越模糊，很多加壳软件除具有较强的压缩性能，同时也有了较强的保护性能。

加壳软件一般都有良好的操作界面，使用也比较简单。除了一些商业壳，还有一些个人开发的壳，种类较多。壳对软件提供了良好保护的同时，也带来了兼容性的问题，选择一款壳保护软件后，要在不同硬件和系统上多测试。由于壳能保护自身代码，因此许多木马或病毒都喜欢用壳来保护和隐藏自己。对于一些流行的壳，杀毒引擎能对目标软件脱壳，再进行病毒检查。而大多数私人壳，杀毒软件不会专门开发解压引擎，而是直接把壳当成木马或病毒处理。

有加壳就一定会有脱壳。一般的脱壳软件多是专门针对某加壳软件而编的，虽然针对性强、效果好，但收集麻烦。因此掌握手动脱壳技术十分必要。

12.1.2　压缩引擎

一些加壳软件能将文件压缩，大多数情况下，压缩算法是调用现成的压缩引擎。目前压缩引擎种类比较多，不同的压缩引擎有不同特点，如一些对图像压缩效果好，一些对数据压缩效果好。而加壳软件选择压缩引擎有一个特点，在保证压缩比的条件下，压缩速度慢些关系不是太大，但解压速度一定要快，这样加了壳的EXE文件运行起来速度才不会受太大的影响。例如下面几个压缩引擎就能满足这样要求：aPLib、JCALG1、LZMA。

12.2　压缩壳

不同的外壳所侧重的方面也不一样，有的侧重于压缩，有的则侧重于加密。压缩壳的特点就是减小软件体积大小，加密保护不是其重点。目前兼容性和稳定性比较好的压缩壳有UPX、ASPack、PECompact等。

12.2.1 UPX

UPX是一个以命令行方式操作的可执行文件免费压缩程序，兼容性和稳定性很好。UPX包含DOS、Linux和Windows等版本，并且开源。官方主页：http://upx.sourceforge.net。

UPX的命令格式为：upx [-123456789dlthVL] [-qvfk] [-o file] file..

UPX早期版本压缩引擎是自己实现的，3.x版本也支持LZMA第三方压缩引擎。UPX除了对目标程序进行压缩外，也可解压缩。UPX的开发近乎完美，它不包含任何反调试或保护策略。另外，UPX保护工具UPXPR、UPX-Scrambler等可修改UPX加壳标志，使UPX自解压缩功能失效。

12.2.2 ASPack

ASPack是一款Win32可执行文件压缩软件，可压缩Win32位可执行文件EXE、DLL、OCX，具有很好的兼容性和稳定性。官方主页：http://www.aspack.com。

双击ASPack运行软件，出现了程序界面（见图12.2）。如果压缩过程中出错，请将选项中的“压缩资源”去除选中。其他加壳软件也会出现类似问题，解决办法一样。

[image]

图12.2　ASPack程序界面

12.3 加密壳

加密壳种类比较多，不同的壳侧重点不同，一些壳单纯保护程序，另一些壳还提供额外的功能，如提供注册机制、使用次数、时间限制等。加密壳还有一个特点，越是有名的加密壳，研究的人也越多，其被脱壳或破解的可能性也越大，所以不要太依赖壳的保护。加密壳在强度与兼容性上做得好的并不多，这里向大家介绍几款常见的。

12.3.1 ASProtect

ASProtect是一款非常强大的Win32位保护工具，这款壳开创了壳的新时代。它拥有压缩、加密、反跟踪代码、CRC校验和花指令等保护措施。它使用Blowfish、Twofish、TEA等强劲的加密算法，还用RSA1024作为注册密钥生成器。它还通过API钩子与加壳的程序进行通信，并且ASProtect为软件开发人员提供SDK，实现加密程序内外结合。SDK支持VC、VB、Delphi等。

ASProtect创建者是俄国人Alexey Solodovnikov，其将ASPack中的一些开发经验运用到ASProtect，该壳的编写简单而精巧，是款经典之作。ASProtect很注重于兼容性和稳定性，因此没有采用过多的反调试策略。

ASProtect目前有两个系列，一个系列是ASProtect 1.3x，另一个系列是ASProtect SKE 2.x。ASProtect SKE系列已采用了部分虚拟机技术，主要是在Protect Original EntryPoint与SDK上。保护过程中建议大量使

用SDK，SDK使用请参考其帮助文档及安装目录下的样例，在使用时注意SDK不要嵌套，并且同一组标签用在同一个子程序段里。

运行ASProtect，主界面如图12.3所示。首先单击“File To Protect”的[image]按钮选择一个需要保护的文件，在“Output To FileName”中填上输出的文件名。再单击“Modes”标签，单击“Add Mode”按钮，将“Is this Mode Avtive”选上。最后，单击“Protection”标签，对软件进行保护即可。

[image]

图12.3　ASProtect主界面

ASProtect加壳过程中也可挂接用户自己写的DLL文件，方法是在图12.3中的“External Options”选项中加上目标DLL即可。这样，用户可以在DLL中加入自己的反跟踪代码，以提高软件的反跟踪能力。

由于ASProtect在共享软件里使用得相当广，因此大家研究得也多，其各类保护机制已被研究得很透了，甚至可能都有脱壳机的存在。

12.3.2　Armadillo

Armadillo也称穿山甲，是一款应用面较广的商业保护软件（其界面如图12.4所示）。可以运用各种手段来保护你的软件，同时也可以为软件加上种种限制，包括时间、次数，启动画面等。其官方主页：http://www.siliconrealms.com。Armadillo对外发行时有Public、Custom两个版本。Public是公开演示的版本，Custom是注册用户拿到的版本。只有Custom才有完整的功能，如强大的Nanomites保护。Public版有功能限制，没什么强度，不建议采用。

[image]

图12.4　Armadillo界面

Armadillo有如下保护功能：Nanomites、Import Table Elimination、Strategic Code Splicing、Memory-Patching Protections等。其中Nanomites功能最为强大，使用时，需要在程序里加入Nanomites标签，如下面这段VC编译器里定义的标签：

[image]

用NANOBEGIN和NANOEND标签将需要保护的代码括住，Armadillo加壳时，会扫描程序，处理标签里的跳转指令，将所有跳转指令换成INT 3指令，其机器码是CC。此时Armadillo运行时，是双进程，子进程遇到CC异常，由父进程截获这个INT 3异常，计算出跳转指令的目标地址并反馈给子进程，子进程继续运行。由于INT 3机器码是CC，因此也称这种保护是CC保护。

12.3.3　EXECryptor

EXECryptor是一款商业保护软件（其主界面如图12.5所示），官方主页：http://www.strongbit.com。其可以为目标软件加上注册机制、时间限制、使用次数等附加功能。这款壳的特点是Anti-Debug比较强大，同时做得比较隐蔽，另外就是采用了虚拟机保护一些关键代码。要使这款壳有强大的保护，必须合理使用SDK功能，将关键的功能代码用虚拟机保护起来。

[image]

图12.5　EXECryptor主界面

12.3.4　Themida

Themida是Oreans的一款商业保护软件（其主界面如图12.6所示），官方链接：www.oreans.com。Themida 1.1以前版本带驱动，稳定性有些影响。Themida最大特点就是其虚拟机保护技术，因此在程序中善用SDK，将关键的代码让Themida用虚拟机保护起来。Themida最大的缺点就是生成的软件有些大。WinLicense这款壳和Themida是同一个公司的一个系列产品，WinLicense主要多了一个协议，可以设定使用时间、运行次数等功能，两者核心保护是一样的。

[image]

图12.6　Themida主界面

12.4　虚拟机保护软件

虚拟机（Virtual Machine，简称VM），现在这个概念已经越来越广泛了。比如VMWare能够在软件环境下模拟出一台单独的机器，VMWare就是一种虚拟机。许多解释性语言，如Visual Basic的P-code程序也是一种虚拟机。这里讨论的虚拟机和VMWare不同，比较类似于P-code，它将一系列的指令解释成bytecode（字节码）放在一个解释引擎中执行，以对软件进行保护。

12.4.1　虚拟机介绍

一个虚拟机引擎由编译器、解释器和VPU Context（虚拟CPU环境）组成，再配上一个或多个指令系统。虚拟机运作的时候，先把已知的x86指令根据自定义的指令系统解释成字节码，放在PE文件中，然后将原处代码删掉，改成如下类似的代码进入虚拟机执行循环。

[image]

从这里可看出，虚拟机保护与加壳保护还是不同的。调试者跟踪进入到虚拟机后，是非常难于理解原指令的。就好像把一篇文章从英文翻译到中文，结果发现文章的很多段落是用孟加拉语写的。

如果分析者要跟踪虚拟机内的代码执行，这将是一件非常繁重的工作。要想理解程序流程，就必须对虚拟机引擎进行深入分析。完整地得

到原始代码和P-code的对应关系，复杂性可想而知了。也就是说，虚拟机加密策略，是建立在提高解密者的分析成本上的一种加密策略。正是由于这个原因，虚拟机已经成为最流行的保护趋势。

虚拟机技术是以效率换安全的，往往一条原始汇编指令经过VM处理后会膨胀几十倍甚至几百倍，执行速度会大大降低。正是由于这个原因，VM保护一般提供SDK方式，使用者一般只需要把较为重要的代码用VM保护起来，这样在一定程度上确保了程序的执行效率。由于当前CPU速度足够快，对于一般程序用虚拟机处理后，不会太多影响程序性能。如果对于一些对速度要求高的代码，就不适合用虚拟机保护了。

现今这一技术逐渐被用于软件保护技术中，如EXECryptor、Themida、VMProtect等商业保护软件。前两者是将壳与虚拟机结合起来了，由于设计原因，其虚拟机强度没有VMProtect强大，VMProtect是一款纯虚拟机保护软件。

12.4.2　VMProtect简介

VMProtect适合Visual Basic(native)、Visual C、Delphi、ASM等本地编译的目标程序，支持EXE、DLL、SYS。VMProtect是由俄国的PolyTech开发，一个利用伪指令虚拟机的保护软件。VMProtect并不是一款壳，它将指定的代码变形（Mutation）和虚拟化（Virtualization），处理后能很好地隐藏代码算法，防止算法被逆向。

1．保护指定代码

VMProtect可以精确地保护指定地址的代码，用调试器（如OllyDbg）跟踪目标程序，找到要保留的核心代码，获得相关地址。然后用VMProtect打开待保护的文件，单击“Project”菜单下的“New procedure”或者单击工具栏中的[image]按钮，在出现的对话框中填入这个地址。也可在Dump窗口，找到待保护的代码，再执行菜单“New procedure”，如图12.7所示。重复这个过程，将其他要保护的地址添加上。每添加一个要保护的地址，VMprotect会根据代码的执行流程判断最可能结束的地址。如果要指定结束地址，在相应地址上，单击鼠标右键，执行“End of procedure”功能。

[image]

图12.7　VMProtect主界面

用调试器获得地址的操作过程比较专业，不适合大众所使用。因此，VMProtect自1.2版本以后，支持SDK。在编程时，用VMProtect提供一对标记将需要保护的代码括住，然后用VMProtect打开编译后的EXE文件时，VMProtect就会认出这些标记，并在有标记的地方进行保护。

在Delphi程序中的SDK。标记开始：

[image]

标记结束：

[image]

在VB程序中的标记。标记开始：

[image]

标记结束：

[image]

在VC中可以利用_emit语句插入十六进制字节代码来实现，具体见本书光盘映像文件中样例。

运行VMProtect，打开带有SDK的目标文件，单击工具栏中的[image]按钮，在弹出的添加地址窗口中会自动将SDK定义代码的地址填上，如图12.8所示。

[image]

图12.8　显示SDK标签处的代码

2．保护函数

VMProtect支持Map文件来定位函数，设置编译器，让其生成Map文件。这个Map文件含有程序中自定义函数和引用外部函数的名称和地址，VMProtect会用这个Map文件来定位函数的地址。将目标文件和Map文件放在一起（文件名要相同），用VMProtect打开文件后，执行菜单“New procedure”时能够列出很多内部函数，这时只需要选择想加密的函数进行后续处理即可，如图12.9所示。如果是目标文件有输出函数，VMProtect还将列出各输出函数的地址。

[image]

图12.9　打开有Map程序的情况

3．加密保护

在VMProtect里确定了将待加密的地址后，单击“Options”标签，根据需要设置相应的选项，如图12.10所示。最后单击菜单“Project/Compilation”（F9），便可对目标软件进行保护。经VMProtect处理好的文件，要多测试，并建议用调试器OllyDbg再检查一下目标代码。经处理过的软件，可以继续用ASProtect、Themida等加壳软件进一步保护，不过笔者认为没有太多的必要了，VMProtect保护的代码安全性已经很好了。

[image]

图12.10　设置VMProtect的选项

VMProtect低版本对多线程程序处理不是太好，因此不要同时处理

多个不同线程的代码。处理时，可以先处理一个线程内的代码，编译后，再用VMProtect处理另一个线程内的代码。测试时，可以到超线程的CPU或双CPU硬件上测试。VMProtect高版本在这方面有所完善。

VMProtect是当前最强的虚拟机保护软件之一，经过VMProtect处理的软件大大增加了逆向人员的分析困难性，关键是用好。另外，经虚拟机处理代码效率会降低，因此一些对效率要求比较高的代码就不适合用VMProtect进行处理。

第13章　脱壳技术

任何事物都有两面性，有加壳，就必有脱壳。加壳与脱壳有着紧密的联系，一些脱壳技术是针对加壳而产生的，脱壳的进步，又迫使加壳软件不断创新发展。现在越来越多的软件都加壳保护，脱壳有时是分析一个软件不可缺少的步骤。

13.1　基础知识

现阶段加壳软件种类比较多，各种反跟踪技术和保护技术都被应用上了，如要脱壳，必须对壳的一些共性原理有所掌握。

13.1.1　壳的加载过程

壳和病毒在某些方面比较类似，都需要比原程序代码更早地获得控制权。壳修改了原程序的执行文件的组织结构，从而能够比原程序的代码提前获得控制权，并且不会影响原程序的正常运行。这里简单说说壳的常见加载过程。

1．保存入口参数

加壳程序初始化时保存各寄存器的值，外壳执行完毕，再恢复各寄存器内容，最后再跳到原程序执行。通常用pushad/popad、pushfd/popfd指令对来保存与恢复现场环境。

2．获取壳自己所需要使用的API地址

一般外壳的输入表中只有GetProcAddress、GetModuleHandle和LoadLibrary这几个API函数，甚至只有Kernel32.dll以及GetProcAddress。如果需要其他的API函数，则通过LoadLibraryA(W)或LoadLibraryExA(W)将DLL文件映像映射到调用进程的地址空间中，函数返回的HINSTANCE值用于标识文件映像映射到的虚拟内存地址。

LoadLibrary函数的原型如下：

[image]

如果DLL文件已被映射到调用进程的地址空间里，可以调用GetModuleHandleA(W)函数获得DLL模块句柄。函数的原型如下：

[image]

一旦DLL模块被加载，线程就可以调用GetProcAddress函数获取输入函数的地址。函数原型如下：

[image]

参数hModule是调用LoadLibrary(Ex)或GetModuleHandle函数的返回值。参数lpProcName可以采用两种形式：第一种是以0结尾的字符串

地址；第二种形式是调用地址的符号的序号（微软公司非常反对使用序号）。

读者必须熟练掌握这三个函数的用法，外壳中用到其他函数就是用这三个函数来调用的。现在有些壳，为了提高强度，甚至连系统提供的GetProcAddress函数都不用，而是自己写个相同功能的函数代替GetProcAddress，以提高函数调用的隐蔽性。

3．解密原程序的各个区块的数据

壳出于保护原程序代码和数据的目的，一般都会加密原程序文件的各个区块。在程序执行时外壳将会对这些区块数据解密，以让程序能正常运行。壳一般是按区块加密的，那么在解密时也按区块解密，并且把解密的区块数据按照区块的定义放在合适的内存位置。

4．IAT的初始化

IAT的填写，本来应该由PE装载器实现。但由于加壳时，自己构造了一个输入表，并让PE头中的输入表指针指向了自建的输入表。所以，PE装载器就将对自建的输入表进行了填写。那么原来PE的输入表的填写，只好由外壳程序实现了。外壳要做的就是将这个新输入表结构从头到尾扫描一遍，对每一个DLL引入的所有函数重新获取地址，并填写在IAT表中。

5．重定位项的处理

文件执行时将被映射到指定内存地址中，这个初始内存地址称为基址。当然这只是程序文件中声明的，程序运行时能够保证系统一定满足其要求吗？

对于EXE的程序文件来说，Windows系统会尽量满足。例如某EXE文件的基地址为400000h，而运行时Windows系统提供给程序的基地址也同样是400000h。在这种情况下就不需要进行地址“重定位”了。由于不需要对EXE文件进行“重定位”，所以加壳软件把原程序文件中用于保存重定位信息的区块干脆也删除了，这样使得加壳后的文件更加小巧。有些工具提供“Wipe Reloc”的功能，其实就是这个作用。

不过对于DLL的动态链接库文件来说，Windows系统没有办法保证每一次DLL运行时提供相同的基地址。这样“重定位”就很重要了，此时壳中也需要提供进行“重定位”的代码，否则原程序中的代码是无法正常运行起来的。从这点来说，加壳的DLL比加壳的EXE修正时多了一个重定位表。

6．HOOK-API

程序文件中的输入表的作用是让Windows系统在程序运行时提供API的实际地址给程序使用。在程序的第一行代码执行之前，Windows系统就完成了这个工作。

壳一般都修改了原程序文件的输入表，然后自己模仿Windows系统的工作来填充输入表中相关的数据。在填充过程中，外壳就可填充HOOK-API的代码的地址，这样就可间接地获得程序的控制权。

7．跳转到程序原入口点（OEP）

从这个时候起壳就把控制权交还给原程序了，一般的壳在这里会有明显的一个“分界线”。当然现在越来越多的加密壳将OEP一段代码搬到外壳的地址空间里，然后将这段代码清除掉。这种技术称为Stolen Bytes。这样，OEP与外壳间就没那条明显的分界线了，增加了脱壳难度。

13.1.2　脱壳机

针对特定的壳，开发出来的脱壳软件，称为脱壳机。脱壳就是将加壳后的程序恢复原来状态，脱壳成功的标志是文件能正常运行。由于脱壳有时没有将壳本身的代码去除，脱壳后的程序一般会比原始程序大。

脱壳机一般分为专用脱壳机和通用脱壳机。专用脱壳机是针对某种壳专门编写的，只能脱特定的壳，使用范围小，但效果好。通用脱壳机则具有通用性，可以脱多种不同类型的壳，主要是压缩壳，例如Quick Unpack、File Scanner等。

分析一个软件前，用PEiD确定一下壳的种类，然后选用合适的脱壳机。脱壳机使用都比较简单，本节就不介绍了。

13.1.3　手动脱壳

对于一些加密壳或修改的壳，没有脱壳机，此时必须要分析外壳，手动脱壳了。手动脱壳可以加深对PE格式的理解，并能从一些外壳里吸取先进的加密技术。手动脱壳过程一般分为三步：一是查找程序的真正入口点；二是抓取内存映像文件；三是PE文件重建。

手动脱壳原理请参考13.1.1节“壳的加载过程”。程序执行时，外壳代码首先获得控制权，模拟Windows加载器，将原来的程序恢复到内存中。这时，内存中的数据就是加壳前的映像文件了，适机抓取出来，再修正一下即可还原到加壳前的状态。

13.2　寻找OEP

外壳保护的程序运行时，首先执行外壳程序，外壳程序负责把用户原来的程序在内存中解压还原，并把控制权交还给解开后的真正程序，再跳到原来程序的入口点，一般的壳在这里会有明显的一个“分界线”，这个解压后程序真正的入口点称为OEP，即Original Entry Point。

13.2.1　根据跨段指令寻找OEP

绝大多数PE加壳程序在被加密的程序中加上一个或多个区块，当外壳代码处理完毕后，会跳到程序本身的代码上，所以依据跨段的转移指令就可找到真正的入口点。

用第16章编写的加壳工具对实例进行加壳处理，加壳后的程序称为RebPE。本节用这个实例来演示如何依据跨段指令寻找OEP。

为了学习方便，先查看一下加壳前程序的一些信息。用LordPE打开加壳前的实例，获得其入口点RVA为1130h。再查看区块，如图13.1所示。

[image]

图13.1　查看加壳前的区块

加壳后RebPE的入口点RVA为13000h。再查看区块，如图13.2所示。

[image]

图13.2　查看加壳后的区块

加壳后，RebPE多了一个区块.pediy，这个区块就是外壳部分，其相当于一个文件加载器（Loader）。RebPE运行时，各区块被Windows系统映射到内存中，现在的入口点是13000h，指向外壳部分。外壳拿到控制权后，通过各种方式获得自己所需要的API地址，解密原程序各区块的数据，填充IAT，做完这些工作后，就准备跳到原程序入口点（OEP），即401130h处执行，如图13.3所示。

[image]

图13.3　外壳加载运行

运行OllyDbg后，在调试选项里，将暂停点设置在主模块的入口点（Entry point of main module）。用OllyDbg打开加壳后的实例RebPE，可能会提示所加载的程序入口点超出代码范围，如图13.4所示。这是因为现在入口点指向外壳部分，即区块.pediy部分，不是常见的代码段（本例是.text区块），所以OllyDbg会给出提示。可以在选项里将这个提示关闭。

[image]

图13.4　提示入口点超出代码范围

OllyDbg暂停后，加壳程序的入口点代码如下（对各句汇编代码的理解，可以参考第16章中的16.3.4节“外壳引导段”）：

[image]

[image]

第二段的主要工作是还原各区块数据，初始化原程序，如填充IAT表。由于外壳的第二段空间是调用VirtualAlloc函数随机分配的，因此读者系统显示的地址和笔者可能会不同，阅读时，请以汇编代码来参考。

[image]

[image]

[image]

[image]

至此，外壳代码处理完毕，其已将目标程序初始化完毕，然后从外壳段直接跳到代码段执行，即用跨段的转移指令跳到真正的入口点执行解压后的程序。转移指令如下：

[image]

这句指令相当于jmp 401130。来到401130处，读者可能会看到如下代码：

[image]

此时按“Ctrl + A”键强迫OllyDbg重新分析一下代码即可。

[image]

再来完整地回顾一下外壳初始化过程。外壳部分用了两次跨段转移指令：一次是从.pediy区块跳到外壳第二段（调用VirtualAlloc随机分配）；第二次是从外壳第二段，跳到程序本身代码所处的区块，本例是.text区块，这也是一般判断是否OEP的关键。停在OEP时，在OllyDbg里按“Alt + M”键，可以看到各模板情况，如图13.5所示。

[image]

图13.5　查看实例的内存模块

这个方法没什么技巧，从壳的开始一直跟踪，直到来到代码段本身，从而确定OEP。

13.2.2　用内存访问断点找OEP

外壳首先将原来压缩的代码解压，并放到对应的区块上，处理完毕，将跳到代码段执行。上一节是手动跟踪这个过程，一直来到代码段上。很高兴，OllyDbg有对代码段设断的功能，可以避免手动一步步地跟踪了。当对代码段设置内存访问断点时，一定会中断在外壳对代码写入的那句上面。

按“Alt + M”键打开内存模板，对代码段（本例是.text区块）按F2键下内存访问断点，如图13.6所示。这个断点是一次性断点，当所在段被读取或执行时就中断，中断发生以后，断点将被自动删除。

[image]

图13.6　对代码段下内存访问断点

对.text区块设置内存访问断点后，按F9键执行程序，程序将中断如下代码处：

[image]

上面这段代码是Aplib解压函数aP_depack_asm，走出这个函数，将来到外壳代码处。具体如下：

[image]

这段代码依次将.text、.rdata、.data、.rsrc区块解压，并放到正确的位置上。此时代码段全部解压完毕后，再对代码段（.text区块）设置内存访问断点，按F9键执行程序，当外壳跳到OEP返回代码段时，即可触发内存访问断点而中断。

这个方法的关键是要等代码段解压完毕，再对代码段设置内存访问断点。如果之前设置断点，会不停地中断在对代码写入的指令上。解决这个问题还有一个更好的方法，即下两次内存断点办法。一般的壳，会依次对.text、.rdata、.data、.rsrc区块进行解压处理，所以，可以先在.rdata、.data等区块下内存访问断点，中断后，此时代码段已解压，接着再对代码段（.text块）下内存访问断点，即可达到OEP。

13.2.3　根据堆栈平衡原理找OEP

编写加壳软件时，必须保证外壳初始化的现场环境（各寄存器值）与原程序的现场环境是相同的（主要是ESP、EBP等重要的寄存器值）。加壳程序初始化时保存各寄存器的值，外壳执行完毕，再恢复各寄存器内容，最后再跳到原程序执行。通常用pushad/popad、pushfd/popfd指令对来保存与恢复现场环境，其中EFLAGS（标志寄存器）影响不是太重要，一般不处理。也就是说，编写加壳软件时，必须遵守堆栈平衡的原理，具体如下：

[image]

脱壳时，可以根据这个堆栈平衡的原理对ESP设断，能很快找到OEP。

用OllyDbg加载RebPE，当执行pushad指令后，各寄存器的值将被压入0012FFA4h～0012FFC0h的堆栈中，如图13.7所示为堆栈窗口。

[image]

图13.7　查看堆栈窗口

此时ESP指向12FFA4h，对这个地址下硬件访问断点，如图13.8所示。

[image]

图13.8　下硬件访问断点

按F9键运行程序，外壳代码处理结束后，调用popad指令恢复现场环境时，访问这些堆栈，OllyDbg将会中断，此处离OEP也不远了。

[image]

可以把整个外壳当做一个函数或子程序来理解，这个过程遵守堆栈平衡的原理，当其跳到OEP时，ESP的值不会变。了解这个现象后，就可以用另一个方法直接来到OEP了。当PE文件开始运行时，记下当时的ESP值，假设是12FFC4h，而大多数程序第一句都是压栈（push指令），而这句就是对12FFC0h进行写入操作，因此对其设置硬件写断点（如图13.9所示），就可方便地来到OEP附近。

[image]

图13.9　下硬件写断点

[image]注意：程序刚开始运行时，ESP的值是由操作系统决定的。

13.2.4　根据编译语言特点找OEP

各类语言编译的文件入口点都有自己的特点，同一种编译器，其入口代码都很类似，都有一段启动代码，编译器编译程序时，自动连接到程序中去。其完成必需的初始化工作后，再调用WinMain函数，执行完后，启动代码再次获得控制权，进行一次初始化清除工作。

例如，对于所有的Visual C＋＋ 6.0程序来说，其默认的入口点代码大概有4个版本，处理ANSI字符的GUI版本，其启动函数是WinMainCRTStartup；处理ANSI字符的CUI版本，其启动函数是mainCRTStartup；还有两个Unicode版本。具体描述参考第4章的“4.1启动函数”这一节。由于VC 6启动部分都有这几个函数，如GetCommandLineA(W)、GetModuleHandleA(W)、GetVersion、GetStartupInfoA(W)，因此可以用来设置断点，很方便定位程序的OEP。

用OllyDbg加载RebPE，对GetVersion函数设置断点。中断两次后，就能返回到OEP附近。

[image]

读者对常见语言的入口代码都要熟悉，在脱壳修复或定位OEP时，将带来很大的方便。

采用默认的启动代码对软件加壳保护不是很有利，一些开发人员于是对启动源代码进行修改，这时程序的入口点与默认的完全不同。

13.3　抓取内存映像

抓取内存映像，也称为转存，英文称为Dump。其实是把内存指定地址的映像文件读取出来，然后用文件等形式保存下来。

脱壳时，在何时Dump文件是有一定技巧的。一般情况下，外壳来到OEP处Dump是正确的。如果程序运行起来再Dump，一些变量已初始化了，不适合再Dump了。在外壳处理过程中，它要把压缩了的全部的代码数据释放到内存中，初始化一些项目，此时也可以选择合适的点Dump。

13.3.1 Dump原理

常用的Dump软件有LordPE、ProcDump、PETools等。这类工具一般是利用Module32Next来获取欲Dump进程的基本信息的。Module32Next函数的原型如下：

[image]

参数：

• hSnapshot：由先前的CreateToolhelp32Snapshot函数返回的快照；

• lpme：指向MODULEENTRY32结构的指针。

每次执行函数后，它都将把一个进程的信息填入MODULEENTRY32结构中。MODULEENTRY32结构的定义如下：

[image]

ProcDump和LordPE都是根据此结构中的modBaseSize和modBaseAddr字段得到进程的映像大小和基址，再调用ReadProcessMemory来读取进程内的数据。读取成功以后，ProcDump会检测IMAGE_DOS_SIGNATURE和IMAGE_NT_SIGNATURE是否完整。如果完整它就基本不再对其他大多数字段做检验了，如果不完整它会根据szExePath字段打开进程的原始文件，读取它的文件头以取代进程的文件头。LordPE则更简单，它根本不用进程的文件头，直接读取原始文件的文件头来用。在读取内存数据后，再把进程中的数据保存到硬盘的文件里。

在这里以LordPE讲述一下此类工具用法。在如图13.10所示的Options里，默认选上“Full dump:paste header from disk”，也就是说，PE头的信息是直接从磁盘文件获得的。

[image]

图13.10　LordPE选项设置

设置好后，在LordPE的进程窗口选择相关进程，单击右键，执行“dump full”菜单命令，即可抓取文件并保存到文件里，如图13.11所示。

[image]

图13.11　LordPE里Dump映像文件

OllyDbg的一个插件OllyDump，也支持Dump功能，使用也比较方便，缺点是不方便Dump取DLL文件的映像。

13.3.2 反Dump技术（Anti-Dump）

Dump是脱壳过程中的一个关键步骤，部分加密外壳会采取Anti-

Dump技术来防止被脱壳，为了能顺利脱壳，就必须绕过这些Anti-Dump。

1．纠正**SizeOfImage**

Dump文件时，一些关键参数是通过MODULEENTRY32结构快照获得的，因此可以在modBaseSize和modBaseAddr字段中填入错误的值，让Dump软件无法正确读取进程的数据。经测试发现，如果修改系统中modBaseAddr的值会让系统出现问题，所以只能修改modBaseSize的值，方法如下：

[image]

在程序中插入这样一段代码后，用Module32Next函数得到的该进程的映像大小就是1000h字节，Dump软件也就只会读出1000h字节的程序数据。对于ProcDump，很可能在进行后续工作时产生非法操作，而用LordPE得到的将只是一个大小为4KB的无用文件。

如何来防止它呢？当然就是找出类似的代码，然后跳过它。LordPE的“correct ImageSize”功能也可以处理这个Anti，其原理是打开磁盘文件，直接读取PE头的SizeOfImage来纠正这个错误。实例RebPE就带有这个Anti-Dump，运行后，在相关进程的右键菜单中执行“correct ImageSize”功能，然后再dump full，可以得到完整的文件，如图13.12所示。

[image]

图13.12　LordPE里“correct ImageSize”功能

2．修改内存属性

当PE文件被加载到内存中时，其所有段的属性都是可读的，这样Dump工具打开进程时，就可读取内存数据并转存到磁盘文件中。如果进程的某个地址不可读，某些Dump工具可能不能正确读取相关数据。

本书光盘映像文件中提供的实例Modify_the_read_right使用VirtualProtect函数将PE头设为不可读，运行后，用LordPE来Dump，会出现如图13.13所示的错误框。

[image]

图13.13　不能读取内存数据出错

需要查看内存指定区域属性的时候，可以使用LordPE的“dump region”功能。选中进程，执行右键菜单中的“dump region”，如图13.14所示。地址400000h处PE头部显示为NOACCESS（不可读写）属性，这样用LordPE工具读取不了数据。

[image]

图13.14　查看内存属性

在这种情况下，如何Dump内存映像呢？可以用OllyDbg加载目标程序，运行后，按“Alt + M”键打开内存映像，在该进程的PE头单击右键，执行“Set access/Full access”，将PE头设置为完整权限，如图13.15所示。这样，再运行LordPE，就可Dump内存映像了。

另一款PE工具——PETools 1.5.8却能成功Dump该实例进程，但是查看抓取出的映像文件，会发现PE头部分全是0。跟踪一下PETools的Dump过程，会发现它遇到NOACCESS页面并没有Dump，直接放弃，它只Dump能读的页面了。也就是说，在这种情况下PETools抓取的内存映像是不完整的。

[image]

图13.15　设置区块属性

13.4　重建输入表

在加密外壳中，破坏原程序的输入表是必有的功能，在脱壳中输入表处理是很关键的一个环节，因此要求脱壳者对PE格式中的输入表概念非常清楚。

13.4.1　输入表重建的原理

输入表结构中与实际运行相关的主要是IAT结构，这个结构用于保存API的实际地址。PE文件运行时，在初始化输入表这块时，Windows装载器首先搜索OriginalFirstThunk，如果存在，加载程序迭代搜索数组中的每个指针，找到每个IMAGE_IMPORT_BY_NAME结构所指向的输入函数的地址，然后加载器用函数真正入口地址来替代由FirstThunk指向的IMAGE_THUNK_DATA数组里的元素值。当初始化结束时，输入表情况如图13.16所示。

[image]

图13.16　PE文件加载后的IAT表

此时输入表中其他部分就不重要了，程序依靠IAT提供的函数地址就可正常运行。对于外壳程序，一般都修改了原程序文件的输入表，然后自己模仿Windows装载器的工作来填充IAT中相关的数据，也就是说，内存中就一张IAT表，原程序的输入表是不存的。输入表重建就是根据这个IAT还原整个输入表的结构（即图13.16所示的这个结构），如IID结构及其各成员指向的数据等。

一些加密软件为了防止输入表被还原，就在IAT加密上大作文章，此时外壳填充IAT里的不是实际的API地址，而是填充壳中用来HOOK-API的外壳代码的地址。这样外壳中的代码一旦完成了加载工作，在进入原程序的代码之后，仍然能够间接地获得程序的控制权。因为程序总是需要与系统打交道，与系统打交道的途径是API，而API的地址已经替换成了外壳的HOOK-API的地址，那每一次程序与系统打交道，都会让外壳的代码获得一次控制权，这样外壳可以进行反跟踪继续保护软件，

同时也可完成某些特殊的任务。所以重建输入表的关键是获得没加密的IAT，一般的做法是跟踪加壳程序对IAT处理过程，修改相关指令，不让外壳加密IAT。

13.4.2　确定IAT的地址和大小

输入表重建的关键就在于IAT的获得，一般程序的IAT是连续排列的，以一个DWORD字的0作为结束，因此只要确定IAT某个点，就能获得整个IAT地址和大小。

程序中每一个API函数在IAT里都有它自己的位置，这样无论代码中多少次调用一个输入函数，都会通过IAT中的同一个函数指针来完成。要确定IAT的地址，先看看程序是怎样调用一个输入函数的。一种情况是像下面这样：

[image]

直接调用[405028]中的函数，地址405028h位于IAT里，指向GetVersion函数。另一种API调用像下面这样：

[image]

在这种情况下，CALL把控制权转到一个子程序，子程序中的JMP指令跳转到位于IAT中的50D330h。

本节通过实例演示一下如何确定IAT位置。用OllyDbg打开没有加壳的RebPE.exe，随便找一句调用API函数的语句，如图13.17所示。

[image]

图13.17　调用API函数的语句

地址405028就是IAT中的一个部分，在数据窗口查看其内容，如图13.18所示。

[image]

图13.18　在数据窗口查看IAT

图13.18所示的每一个DWORD数据都是指向一个API函数，如FA 11 81 7C就是地址：7C8111FA，在OllyDbg里按“Ctrl + G”键，输入7C8111FA跳到这个地址就会发现是GetVersion函数，如图13.19所示。

[image]

图13.19　查看API函数

IAT是一块连续排列的数据，因此在数据窗口向上翻屏，直到出现00数据，寻找IAT起始地址，本例翻屏后直到405000h，此处也是.rdata区块的起始端，如图13.20所示。

[image]

图13.20　确定IAT起始地址

再向下翻屏确定IAT末端，输入表每个DLL对应一个IAT，一般这些IAT以一个数据为0的DWORD间隔。IAT最后是以0结尾的，如图13.21所示，4050B8h处就是IAT结尾。也就是说，IAT的地址是405000h，大小为4050B8h-405000h = B8h。

[image]

图13.21　确定IAT结束地址

为了直观些，也可以这样让数据窗口直接显示这些API函数，以确定IAT是否正确，在数据窗口单击右键，执行菜单“Long/Address”命令，如图13.22所示。

[image]

图13.22　切换数据窗口显示模式

调整后的数据窗口就直观了，直接显示调用的API函数名，如图13.23所示。若要恢复原有模式，执行菜单“Hex/ASCII(16 bytes)”命令。

[image]

图13.23　以Address模式显示

13.4.3　根据IAT重建输入表

本节演示一下如何利用IAT重建一份完整的输入表，以加深对输入表的理解。当然实际工作中不需要手工构造输入表，可以用ImportREC等专业的输入表重建工具来完成这些工作。

实例Reb_IT.exe输入表比较简单，容易手工构建输入表，用UPX对实例进行加壳处理。用OllyDbg加载已加壳的目标后，在代码窗口中一直往下翻屏，能发现如下代码：

[image]

处理完数据后，UPX外壳用了一次跨段的转移指令（JMP）跳到OEP，会发现OEP为401000h。因此，只需要在4052BFh处设断，中断后，运行LordPE将内存数据Dump出来，保存为dumped.exe。

用上一节方法确定IAT的位置，如图13.24所示。

[image]

图13.24　查看IAT

为了分析方便，用Address模式查看IAT，如图13.25所示。

[image]

图13.25　以Address模式显示IAT

这个程序输入表里输入了两个DLL：一个是kernel32.dll，另一个是user32.dll，分别对应了一份IAT，两份IAT以一个DWORD的0隔开。将IAT结果总结在表13-1中。

表13-1　IAT成员指向的函数名

[image]

用十六进制工具在dumped.exe文件中找一块空间，在这里选择2100h，将表13-1中的DLL名和函数名写进去（见图13.26）。DLL与函数名的位置可以任意。每个函数名前要留两个字节放函数的序号，序号可以为0；每个函数名后的一个字节为0；每个函数名或DLL名起始地址必须按偶数对齐，空隙填充0。

[image]

图13.26　填充IMAGE_IMPORT_BY_NAME结构

由于是内存映像文件，因此文件偏移地址与相对虚拟地址（RVA）值是相等的。将图13.26中各DLL名和函数名所在的偏移地址归纳到表13-2中。

表13-2　各字符串的地址

[image]

[image]注意：下面会忽略所有的文件偏移地址和RVA注释。

根据表13-2构造指向函数名地址的IMAGE_THUNK_DATA数组，具体见表13-3。

表13-3　IMAGE_THUNK_DATA数组

[image]

选择20E0h放IMAGE_THUNK_DATA数组，两个数组之间空2个字节，填充0，如图13.27所示。

[image]

图13.27　填充IMAGE_THUNK_DATA数组

最后构建其IID数组，如表13-4所示。

表13-4　IID数组

[image]

IID数组位置也可以任意，在这里放在2010h地址处。其中IAT位置很重要，不能改变，否则相关指令就找不到函数调用地址（除非再修正这些函数调用的地址），IAT中的内容可以不重新构造，PE文件加载时，Windows系统会填充。在这里将FirstThunk指向原来的IAT，并将OrignalFirstThunk指向的数据复制到FirstThunk指向的空间。图13.28所示是第一个IID结构示意图。

[image]

图13.28　第一个IID结构示意图

图13.29所示是手动构建的完整IID，其中输入表的地址是2010h，大小是28h。

[image]

图13.29　手动构建的IID

输入表的相对虚拟地址（RVA）存储在PE文件头部的目录表中（它的偏移量为[PE文件头偏移量 + 80h]）。

[image]

在130h处写入输入表的地址和大小：1020000028000000；或直接用PE编辑工具修改目录表里的输入表选项，如图13.30所示。

[image]

图13.30　LordPE中修改输入表的地址

13.4.4　ImportREC重建输入表

ImportREC是目前最好用的输入表重建专业工具，它可从杂乱的IAT中重建一个新的输入表，原理参考上一节的手动构造输入表。

1．基本用法

在运行ImportREC之前，必须满足如下条件：

- 目标文件已完全被Dump，另存为一个文件；
- 目标文件必须正在运行中；
- 事先找到目标程序真正的入口点（OEP）或IAT的偏移与大小。

这里以13.2.1节的RebPE.exe为例，讲述ImportREC的使用。让OllyDbg暂停在OEP 401130h处，运行LordPE，由于这个实例有Anti-Dump，执行“correct ImageSize”功能，然后再“dump full”，保存为dumped.exe（此时，一些杀毒软件会提示dumped.exe为病毒，不必理

会）。

准备工作做好，然后运行ImportREC，操作如下：

① 运行ImportREC，在下拉列表框中选择RebPE.exe进程，如图13.31所示。

[image]

图13.31　ImportREC进程下拉列表

② 如有正确的OEP值，则在左下角OEP处填入OEP的RVA值，这里填上1130。默认时，ImportREC重建输入表时会同时用此值修正入口点（选项里可设置），同时提供正确的OEP有助于分析IAT的准确位置。单击“IAT AutoSearch”按钮，让其自动检测IAT偏移和大小，如果出现“Found address which may be in the Original IAT.Try 'Get Import'”对话框，如图13.32所示，这表示输入的OEP发挥作用了。接下来跳到第④步。

[image]

图13.32　提示发现IAT的地址和大小

③ 如没正确的OEP或ImportREC没找到IAT偏移，则手工填入IAT的RVA和大小，如图13.33所示。

[image]

图13.33　IAT的RVA与Size域

④ 单击“Get Imports”按钮，让其分析IAT结构得到基本信息，如图13.34所示。

[image]

图13.34　分析IAT获取输入表相关信息

⑤ 本例所有API函数都能正确识别。如果不能识别，会显示“valid :NO”，单击“Show Invalid”按钮分析所有的无效信息，在“Imported Functions Found”栏中单击鼠标右键，选择“Trace Levell(Disasm)”，再单击“Show Invalid”按钮。如果成功，可以看到所有的DLL都为“valid:YES”。如果仍有无效的地址，可以尝试右键菜单中的“Trace Level 2(HOOK)”或“Trace Level 2(Trap Flag)”命令修复（尽量关闭其他程序）。“Auto Trace”按钮用来自动执行Trace Level 1、Trace Level 2等。

⑥ 最后修复已脱壳的程序Dump.exe。选择“Add new section”（默认为选中）为Dump.exe文件加一个区块，区块名为.mackt（虽然文件变大，但避免了许多不必要的麻烦）。单击“Fix Dump”按钮，并选择刚抓取的映像文件Dump.exe，在此不必备份。如果修复的文件名是“Dump.exe”，它将创建一个“Dump_.exe”，此外OEP也被自动修正。

⑦ 第⑥步脱壳后，输入表放在新增的.mackt区块上。也可以将新的输入表放到程序空白处，本例可以在.rdata选一空白，此处为40545Ch，在ImportREC里填上RVA545Ch（如图13.35所示），再单击“Fix Dump”按钮即可。

[image]

图13.35　指定输入表的存放地址

2．处理不连续的IAT

输入表里，一个DLL对应一份IAT，多个IAT之间一般以一个DWORD的0隔开。也有些程序其IAT被分割成几部分，如Borland C＋＋1999、Borland C＋＋ Builder等程序。

OllyDbg打开实例TestWin.exe，分析获得kernel32.dll所在的IAT为40E0ECh～40E188h，大小为9Ch；gdi32.dll所在的IAT为40E190～40E198h，大小为8h；user32.dll所在的IAT为40E1ECh～40E230h，大小为44h。这三份IAT不连续，各有一段间隙，如图13.36所示。

[image]

图13.36　Borland C程序的IAT排列不连续

遇到这种情况，ImportREC的“IAT AutoSearch”只能自动检测第一份IAT偏移和大小。要得到正确结果，必须依次将其他各小块IAT的地址和大小填进RVA和Size域中，单击“Get Imports”按钮分析IAT结构。重复这个过程，ImportREC会自动将得到的IAT数据组合成一个整体。但这种方法不是很好，也可以直接将整个IAT大小填进Size域中，此处填144h（保证填入的值大于真实Size即可），单击“Get Imports”按钮分析IAT结构。单击“Show Invalid”按钮分析所有的无效信息，在无效信息中单击右键，执行“Cut thunk”功能将无用信息清除，即可得到一份正确的输入表，如图13.37所示。

[image]

图13.37　消除无用的信息

3．修复函数

ImportREC会对个别函数识别出错，这时必须通过跟踪原程序，获得正确的函数，并进行修复。实例apitest.exe是一个简单的程序，在Windows XP系统上，运行ImportREC获得输入表，如图13.38所示。单击“Fix Dump”按钮生成新的apitest_.exe程序。

[image]

图13.38　查看树结构的输入表

修复后的apitest_.exe在Windows XP下能运行，但在Windows 2000

下运行会出现如图13.39所示的错误对话框。

[image]

图13.39　Windows 2000下运行出错

原来在Windows XP下修复输入表的时候，ImportREC会将ntdll!RtlRestoreLastWin32Error重定位到kernel32!RestoreLastError，这个函数在以前版本的Windows系统下是不存在的，这样会造成修复的程序不能跨平台运行。而RestoreLastError和SetLastError实际上是同一个函数，ImportREC将SetLastError识别成RestoreLastError了。

在图13.38所示的kernel32.dll的子节点RestoreLastError上双击鼠标左键，在弹出的输入函数编辑框中对这个输入函数的名字进行修正，改成SetLastError函数，如图13.40所示。这个问题是ImportREC的bug，一些修改版的ImportREC已修正了这个问题。

[image]

图13.40　输入函数编辑框

一般遇到函数识别出错的问题时，解决方法就是在出错函数节点上单击右键，执行“Disassemble/Hex View”查看对应的反汇编代码，将其与没脱壳的程序对比，分析出真实的API，然后再重新构建输入表。

4．其他说明

• 在“Imported Functions Found”栏中单击鼠标右键，可以选择“Expand all nodes”及“Collapse all nodes”来打开/关闭所有节点。

• “Save Tree”将当前输入表以文本文件保存，“Load Tree”从磁盘中导入输入表文本文件。

• 如果IAT自动搜索失败，请尝试如下两种方法：

[image]调整“Options（选项）”里的“Max Recursion（最大循环）”和“Buffer Size（缓冲区大小）”。

[image]跟踪程序，获得IAT的地址和大小。

• 如果IAT被分割成几部分，依次将各小块IAT的地址和大小填进RVA和Size域中，单击“Get Imports”按钮分析IAT结构。重复这个过程，ImportREC会自动将得到的IAT数据组合成一个整体。

• 处理某些壳时，最好是在OEP处将被加壳的进程Suspend（挂起），然后用ImportREC分析。因为某些外壳程序在进入OEP之后会修改IAT的某些项。

• Options的“New Imports”几个选项可以控制新建输入表的一些结构：

[image]“Rebuild OriginalFT”选项就是重建OriginalFirstThunk，否则以0填充，工作时靠FirstThunk。

[image]“Create New IAT”选项是为IAT选定一个新的地址，同时修正代码中对API函数的调用，所以一般不建议勾选。

[image]“Import all by Ordinal”选项表示按序号构建输入表，按序

号构建在不同平台容易出兼容性问题，不推荐使用。

13.4.5　输入表加密概括

脱壳过程，输入表修复是个重点。修复的关键是得到没加密的IAT，这可以对IAT相关的点设断，以找到外壳处理IAT的代码，然后找对策。加壳程序处理输入表有以下几种情况。

（1）完整地保留了原输入表，外壳加载时没对IAT加密

外壳解压数据时，完整的输入表会在内存中出现，然后外壳用显式装载DLL方式获得各函数地址（如GetProcAddress函数），并将该地址填充到IAT中。

脱壳时可以在内存映像文件刚生成时抓取，此时，外壳还没来得及破坏原始的输入表。此类壳有ASPack、PECompact等。

（2）完整地保留了原输入表，外壳装载时对IAT进行加密处理

外壳解压数据时，完整的输入表会在内存中出现，然后外壳用显式装载DLL方式获得各函数地址，并对该地址进行处理（即HOOK-API），最后将HOOK-API的外壳代码的地址填充到IAT。

由于IAT已加密，如直接用ImportREC是重建不了输入表的。但可以在外壳还没来得及加密IAT时，抓取输入表；或者跳过对IAT加密的代码。此类壳有tElock等。

（3）加壳时破坏了原输入表，外壳装载时没对IAT进行加密处理

外壳已完全破坏原输入表，外壳刚解压的映像文件中存在的是输入函数的字符串，然后外壳用显示装载DLL方式获得这些函数的地址，直接将函数地址填充到IAT里。

由于IAT没加密，脱壳时用ImportREC根据IAT重建一个新的输入表。此类壳有UPX等。

（4）加壳时破坏了原输入表，外壳装载时对IAT进行了加密处理

外壳已完全破坏了原输入表，然后外壳用显式装载DLL方式获得各函数地址，并对该地址进行处理（即HOOK-API），最后将HOOK-API的外壳代码的地址填充到IAT。

脱壳时可以利用ImportREC的一些插件对付这些加密的IAT；也可修改外壳处理输入函数地址的代码，让其生成的IAT没加密，然后再用ImportREC重建。此类壳有ASProtect等。

13.5　DLL文件脱壳

DLL是Dynamic Link Library（动态链接库）的缩写形式，是一个共享函数库的可执行文件。DLL文件的脱壳与EXE文件步骤差不多，DLL文件多了个基址重定位表等要考虑。

13.5.1　寻找OEP

当DLL被初次映射到进程的地址空间中时，系统将调用DllMain函数，当卸载DLL时也会再次调用DllMain函数。也就是说，DLL文件相比EXE文件运行有一些特殊性，EXE的入口点只在开始时执行一次，而DLL的入口点在整个执行过程中至少要执行两次。一次是在开始时，用来对DLL做一些初始化。至少还有一次是在退出时，用来清理DLL再退出。

外壳编写时也必须考虑这个因素，初次载入时，做些初始化工作，如IAT初始化等。退出时再次进入入口点时，外壳将跳过相关的初始化代码，这时程序代码流程会短些。所以找OEP也有两条路可以走，一是载入时找，二是在退出时找，退出时流程短些，相对来说更容易找到OEP。

用第16章编写的加壳工具对实例EdrLib.dll进行加壳处理。用LordPE查看其PE信息，获得EntryPoint为D000h，ImageBase为400000h，区块的信息如图13.41所示。

[image]

图13.41　查看区块信息

DLL本身不能直接执行，但可以调用LoadLibrary将DLL的文件映像映射到调用进程的地址空间中，退出时调用FreeLibrary卸载DLL。为了调试DLL，OllyDbg提供了一个类似原理的辅助程序loaddll.exe，这个程序被压缩存放在资源段里，如果OllyDbg所在文件夹内没有loaddll.exe，则会释放这个文件。用OllyDbg打开DLL，将会询问启动loaddll.exe，如图13.42所示。然后链接库被加载并停在程序的入口（ModuleEntryPoint），现在就可以正常调试DLL程序了。

[image]

图13.42　提示是否启动loaddll.exe

OllyDbg加载EdrLib.dll将停在外壳代码第一行。细心的读者会发现，此时EdrLib.dll并没有被映射到默认的400000h内存地址中，而是选择了另一个地址。按“Alt + M”键打开内存映像窗口，将会发现EdrLib.dll被映射到地址E20000h，如图13.43所示。

[image]

图13.43　查看DLL被映射的地址

[image]注意：由于DLL被映射的地址是由系统动态分配的，因此在读者系统中显示的地址可能与本书不同，操作时以当前系统基址为准。下面将忽略所有基地址注释，读者操作时以实际的映像地址来操作。

外壳的入口代码如下：

[image]

此时可以按正常的思路跟踪代码寻找OEP，由于DLL退出时也会来到入口一次，本例演示一下DLL退出时寻找OEP的过程。先在外壳的入口00E2D000处按F2键设置一个断点，按F9键数次让DLL跑起来，DLL装载成功后，loaddll.exe将出现如图13.44所示的界面。

[image]

图13.44　加载DLL成功后的界面

然后将如图13.44所示的窗口关闭，DLL将被卸载，将再次中断在外壳的入口点处。

[image]

来到外壳代码的第二段，这段是解压还原、初始化原程序，为避免重复初始化，第二次进入入口点后将会跳过这些初始化代码。

[image]

跳过外壳初始化代码后，将直接到OEP处。

[image]

此时，OEP的RVA值 = E21240 h - 映像地址 = E21240 h - E20000h = 1240h。

[image]技巧：一般加壳软件用同一套外壳代码处理EXE和DLL，因此在处理EXE文件时，外壳里也有判断是不是多次加载的代码，找到这些跳转，强行改变，这样程序虽不能运行，但能很快定位到OEP相关的代码。

OllyDbg加载某些外壳的DLL文件，不能正常地暂停在外壳的入口点，会直接运行起来。碰到这种情况，可以将外壳的入口点改成死循环，机器码是EBFEh。一个示例如下：

[image]

OllyDbg加载修改后的DLL，由于入口死循环，OllyDbg的CPU窗口显示白屏，按F12键让OllyDbg暂停即可看到代码，再将入口代码恢复成原指令，又可单步调试了。详细操作见光盘映像文件中动画演示。

13.5.2　Dump映像文件

停在OEP后，运行LordPE，在进程窗口中选择loaddll.exe进程，在下方窗口中的EdrLib.dll模块上单击右键，执行“dump full”菜单命令，将文件抓取并保存到文件里，如图13.45所示。

[image]

图13.45　抓取DLL内存映像

1．重定位后抓取映像

对于DLL文件来说，Windows系统没有办法保证每一次运行时提供相同的基地址。如果DLL基址所在内存空间被占用或该区域不够大，系统会寻找另一个地址空间的区域来映射DLL，此时外壳将对DLL执行某些重定位操作。从图13.43得知，此时DLL被映射到内存的地址是E20000h，与EdrLib.dll默认的基址400000h不同，被重定位项所指向的

地方是已经重定位了的代码数据。

如果没有被重定位，一条访问内存的语句应是这样的：

[image]

重定位后，直接访问内存的地址被修正了，语句变成如下：

[image]

为了保证重定位后脱壳后的程序能正常运行，必须修正基址为当前环境的值，本例就是E20000h（见图13.46）。如果被重定位的基址小于默认基址，不可用此法。

[image]

图13.46　修正ImageBase

2．重定位前抓取映像

上一节修改脱壳文件的基址虽也能解决问题，但问题解决得不是很完美，如果脱壳文件的代码和加壳前一样就完美了。为了得到与加壳前一样的文件，可以在DLL载入时跟进外壳，找到重定位的代码，跳过它，让DLL不被重定位。

[image]

在3E01FEh这句强行跳转，不让外壳对代码进行重定位，来到OEP后，就可直接抓取内存映像了。本例中，由于此时代码段数据已完全恢复，也可以在此处直接抓取内存映像。

本例外壳代码较短，单步跟踪能很快找到重定位处理代码，如果外壳比较复杂，就得花点技巧找到重定位处理代码。首先得选取一个重定位的数据，直接对内存地址操作的指令肯定需要重定位。来到OEP后，分析一下相关代码，选取一句会被重定位的语句。例如：

[image]

E21255h这个地址处的数据E2AD68h是被重定位了的。因此，重新加载EdrLib.dll，等代码段解压完毕，在数据窗口对E21253h设内存写断点，即可中断在重定位表处理的代码上。

还可用下两次内存访问断点来寻找重定位处理代码。本实例会依次对.text、rdata、.data、.rsrc区块进行解压处理，所以，可以先在.rdata等区块下内存访问断点，中断后，此时代码段已解压，接着再对代码段（.text块）下内存访问断点，当外壳对代码点进行重定位操作时，即可中断。

13.5.3　重建DLL的输入表

ImportREC能很好地支持DLL的输入表的重建，首先，在Options里将“Use PE Header From Disk”默认的选项去除选中。这是因为

ImportREC需要获得基址计算RVA值，DLL加载的地址不是默认基址时，从磁盘取默认基址计算会导致结果错误。

• 在ImportREC下拉列表框中选择DLL装载器的进程，此处为loaddll.exe进程。

• 单击“Pick DLL”按钮，在DLL进程列表中选择EdrLib.dll进程（见图13.47）。

[image]

图13.47　选择DLL进程

• 在OEP处，填上DLL入口的RVA值1240h，单击“IAT AutoSearch”按钮获取IAT地址。如果失败，必须手工判断DLL的IAT位置和大小，其RVA为7000h，Size为E8h。

• 单击“Get Imports”按钮，让其分析IAT结构重建输入表。

• 勾选“Add new section”，单击“Fix Dump”按钮，并选择刚抓取的映像文件dumped.dll，它将创建一个dumped_.dll文件。

13.5.4　构造重定位表

对于DLL的动态链接库文件来说，重定位数据一般是必需的。外壳很可能破坏了原始的重定位表，将原始的重定位表换个形式存储，运行时模拟PE加载器的重定位功能，对相关代码重定位。脱壳必须根据PE文档的重定位表的定义，重新构造一份新的重定位表。

先来回顾一下重定位表的结构：

[image]

重定位表以1000h大小为一个段，因为ItemOffset最长为12位，即刚好为1000h。如果还有更多段，将重复上面数据结构，直到VirtualAddress为NULL，表示结束。如图13.48所示是重定位表结构的一个示意图。

[image]

图13.48　重定位表结构示意图

外壳重定位相关数据时，会根据外壳转储的重定位表确定要重定位的RVA，再加上当前的基址，完成代码重定位工作。本节构造重定位原理就是将这些要重定位的RVA提取出来，再将这些RVA根据重定位表的定义重新生成一份新的重定位表。根据这个原理，笔者写了一款工具来完成这个重建功能，详见光盘映像文件中的ReloREC。

OllyDbg加载EdrLib.dll，来到重定位初始化的地方，代码如下：

[image]

接下来在上面代码中找到一个点，将需要重定位的RVA取出来，经分析，3E022F这个点比较合适，执行到这句，EBX寄存器保存的就是需

要重定位的地址。补丁的思路是找块代码空间，跳过去执行补丁代码，补丁代码可能是将重定位的地址转成RVA，并保存下来。本例选取3E0289这个地址放补丁，注意此处并不是代码空白处，是存放了外壳的一些参数，当执行3E01FC这句后，这段数据外壳已不用了。因此当OllyDbg第一次来到3E022F这句时，键入如下的补丁指令：

[image]

然后在3E0289h处键入如下补丁代码：

[image]

这段补丁代码读者必须根据本机情况调整一些参数，例如E20000h这是外壳被加载后的基址，3F0000h这个地址是OllyDbg的插件HideOD分配的，如图13.49所示。

[image]

图13.49　利用OllyDbg插件分配临时空间

这个分配空间的地址是随机的，读者系统环境可能会分配到其他值。在3F0000h地址处键入3F0010h，这个地址用来存放获得的重定位RVA，如图13.50所示。

[image]

图13.50　利用一个全局变量当地址指针

补丁代码键入完成后，外壳在处理重定位相关代码时，这段补丁代码将需要重定位的RVA全部提取出来，执行后效果如图13.51所示。

[image]

图13.51　获得需要重定位地址的RVA

从3F0014h开始就是需要重定位代码的RVA，每个地址占用一个DWORD字节。切换到OllyDbg的数据窗口，将3F0014h～3F0AF8h这段需要重定位的RVA复制出来（选取数据时，最后一个DWORD数据是0），操作时单击鼠标右键，执行菜单“Binary/Binary copy”（二进制复制）功能，再运行WinHex，新建一文档，将这段二进制数据粘贴进去，粘贴时，选择“ASCII Hex”模式（见图13.52），然后将提取的数据保存为Relo.bin。Relo.bin中就是需要重定位的地址，以RVA表示。部分数据如下：

[image]

[image]

图13.52　WinHex里以ASCII Hex粘贴

ReloREC这款工具，就是根据这些RVA重新生成一份新的重定位表。准备工作完成后，运行ReloREC，将Relo.bin拖放到ReloREC主界面上可打开此文件，如图13.53所示。在“Relocation's RVA”域里填上原始重定位表的RVA地址，本例为C000h，最后单击“Fix Dump”按钮，打开上节刚修复输入表的dumped_.dll文件，即可完成重定位表的修复。

[image]

图13.53　ReloREC工具界面

13.6　附加数据

某些特殊的PE文件在各个区块的正式数据之后还有一些数据，这些数据不属于任何区块。由于PE文件被映射到内存是按区块映射的，因此这些数据是不能被映射到内存中的，这些额外的数据称为附加数据（overlay）。

附加数据的起点可以认为是最后一个区块的末尾，终点是文件末尾。用LordPE查看实例overlay.exe的区块，如图13.54所示。

[image]

图13.54　查看区块信息

从图13.54可以计算出最后一个区块末尾的文件偏移值为3200h + 600h = 3800h。用十六进制工具打开目标文件，跳到3800h，会发现后面还有一段数据，这就是附加数据，如图13.55所示。

[image]

图13.55　附加数据

用PEiD分析实例overlay.exe，会给出结果“Nothing found [Overlay] *”，其中Overlay就表明有附加数据存在。带有附加数据的文件脱壳时，必须将附加数据粘贴回去，如果文件有访问附加数据的指针，也要修正。

本节实例overlay.exe实际是用UPX加壳了，由于附加数据的存在，干扰了PEiD分析。用OllyDbg打开实例，来到OEP处。

[image]

此时，抓取内存映像保存到磁盘中，然后用ImportREC重建输入表，最终文件为dumped_.exe。

运行实例原文件，然后单击菜单“File/Open”，程序将会读取附加数据并在编辑框中显示出来，如图13.56所示。而运行脱壳后的文件dumped_.exe，不能将原来的文字显示出来，如图13.57所示。

[image]

图13.56　读取附加数据

[image]

图13.57　脱壳后读取附加数据

由于附加数据没有被映射到内存里，因此抓取的映像文件里也没有附加数据。现在将原文件的附加数据移到脱壳后的文件里。用十六进制工具打开overlay.exe，将3800h后的附加数据追加到dumped_.exe文件末尾E000h处。

运行已有附加数据的dumped_.exe，但执行“File/Open”仍不能正确读取数据。用OllyDbg分析一下实例是如何读取自身附加数据的。用CreateFileA设断，执行“File/Open”功能后，会中断到这段代码处。

[image]

用CreateFileA打开一个文件后，文件指针默认是指向文件的第一个字节的。程序用SetFilePointer设置指针，指向附加数据，然后用ReadFile将附加数据读取出来。这里SetFilePointer函数比较关键，其原型如下：

[image]

由于脱壳后，文件大小发生变化，追加后的附加数据地址已改变，此处变为E000h，因此需要修正SetFilePointer的参数，将其指向附加数据，。

[image]

也就是说，对于带有附加数据的程序，抓取内存映像后，必须将附加数据追加到脱壳文件的最后，同时修正读取附加数据的相应指针。

13.7　PE文件的优化

一般脱壳没有将外壳本身的代码去除，资源也没有完全释放，此时脱壳后的程序可能会比原始程序大。虽然脱壳后的文件能正常运行，但在汉化一些场合下可能会遇到问题，比如一些汉化工具不识别脱壳后的文件，或某些功能无效等。本节继续通过实例RebPE来讲解手工优化文件。

1．优化输入表存放位置

OllyDbg加载实例RebPE，来到OEP后，运行LordPE将内存映像抓取出来，另存为dumped.exe。运行LordPE查看dumped.exe的区块信息，如图13.58所示。

[image]

图13.58　查看区块

接下来的步骤一般是用ImportREC重建输入表，默认是将生成的输入表存放在新增的区块上。要使脱壳完美些，尽可能将新输入表存放在原输入表地址处，这需要读者熟悉常见编译器的输入表存放位置，本例是Visual C＋＋ 6.0 SDK编译的程序，输入表一般存放在.rdata区块上。用十六进制工具查看dumped.exe文件的.rdata内容，选取一地址以存放输入表，这个地址必须以DWORD对齐。经查看，发现40545Ch开始有一大段空白，其空间大小可以存放新的输入表，并且VC编译器生成的.rdata区块内容是只读，因此不用担心当程序运行时，这段空白会有数据写入。运行ImportREC，获取输入表后，在新建输入表信息中RVA填545C，如图13.59所示。最后单击“Fix Dump”按钮，选择dumped.exe进行修复，得到dumped_.exe。

[image]

图13.59　指定输入表的存放地址

2．资源的重建

有些软件脱壳后的资源不可查看、编辑或能编辑但保存不了，这是因为脱壳后，资源没完全释放。在“16.2.7资源数据处理”这节中描述了外壳如何处理特殊资源，如Icon（图标）、Group icon（组图标）等，这些图标在程序没有被执行时仍然会被系统读取，它们一般是不能压缩的，所以被存放在外壳本身的代码空间里。正常脱壳后，资源段的其他数据已恢复，但图标等资源还留在外壳里。资源重建，就是把这些资源移回.rsrc区块里。

实例RebPE.exe脱壳后并修复输入表后的文件为dumped_.exe，用ResFixer打开dumped_.exe文件，如图13.60所示。图中显示为红的资源部分位于外壳代码里，如Icon、Group icon都在.pediy区块里。现在，把这些资源移回到.rsrc区块里，这类资源修复工具有多种，常用的有DT_ResFix、freeRes等。

[image]

图13.60　查看资源

运行DT_ResFix，打开dumped_.exe，单击“Fix Resource”按钮，它将分布在多个节里的资源重新移到一个资源节里，并且对资源进行修复优化。重新建立后的资源，用其他资源编辑工具（eXeScope等）就可正常处理了。也可单击“Dump”标签，将重建的资源模块提取出来，如图13.61所示。在“Res File”框里设置好生成文件的路径及文件名，在“NewRVA”框中设置新资源段的RVA，本例是取.rsrc区块的RVA，“FileAligment”对齐值填1000h。设置好后单击“Dump Resource”按钮来Dump资源，文件被保存为rsrc.bin。

[image]

图13.61　重建资源

3．装配文件

现在进行区块调整，PE编辑工具选择LordPE，使用前设置一下LordPE的Options，如图13.62所示。将“Section Table:autofix SizeOfImage”选上，这个功能是自动修正SizeOfImage大小，在增减区块时比较方便。但其自动纠正功能，偶尔也会给使用带来不便，如用其打开某些驱动文件时，LordPE会擅自纠正其SizeOfImage值，这会导致文件的Checksum校验和不正确，从而系统认为驱动文件损坏。

设置好后，用LordPE的PE编辑器打开dumped_.exe，单击“Sections”按钮，进入区块编辑功能中。.pediy区块是外壳的代码，脱壳后不需要，可以删除。.rsrc区块是资源所在区块，上一节已重建新的资源数据，可以删除。在要删除的区块上单击右键，执行“Wipe section header”功能删除区块，但这种删除仅是删除PE头中的数据，对区块的具体内容再用相关的十六进制工具删除。另一款PE工具CFF Explorer能自动删除相关数据，比较方便。

[image]

图13.62　LordPE的选项设置

删除.pediy和.rsrc区块及相应数据后，单击右键，执行“Load section from disk”功能，选中新的资源文件rsrc.bin，将其导入，优化后的区块如图13.63所示。

[image]

图13.63　装配后的区块信息

4．修正PE文件头

用LordPE查看PE头，几个重要的PE字段要修复，主要是EntryPoint、BaseOfCode和BaseOfData，如图13.64所示。

[image]

图13.64　修正PE头信息

- EntryPoint：即脱壳时的OEP，一般ImportREC会自动修正。
- BaseOfCode：代码段的起始RVA。一般是第一个区段（本例是.text段）的RVA，所以这里应该填1000h。
- BaseOfData：数据段的起始RVA。一般指除了代码外的部分开始的RVA，本例就是.rdata区块的RVA5000h。
- SizeOfImage：指装入文件从基址到最后一个块的大小，最后一个块根据其大小往上取整。一般工具会自动纠正这个值。

也可用LordPE的Rebuild PE功能重建程序，偶尔情况下，重建后用来汉化可能会出错。另外，一些优化工具，如PE Optimizer也可选用。

13.8　压缩壳

压缩壳以减小文件体积为目标，加密保护方面不是它的重点，因此生成的IAT都是没加密的，用ImportREC可轻易重建其输入表，如ASPack、UPX等。本节以手动方式探讨这几种壳的调试技巧，如获得OEP、修复输入表、修复重定位表等。目标软件采用DLL文件，相比EXE文件多了一个重定位表需要处理。虽有许多工具可以直接脱壳，但跟踪壳的处理过程才能真正学到本领。

13.8.1　UPX外壳

UPX外壳可以使用UPX自身来去除，这样壳脱得最完美。操作时，使用与加壳所用版本相同的UPX脱壳，或选用更高版本的UPX。

脱壳命令是：

[image]

为了阻止UPX脱其本身的壳，一些保护工具UPXPR和UPX-Scrambler对加壳文件进行处理，使得UPX -d命令失效。解决方法是恢复被破坏的文件或手动脱壳。

1．UPXPR保护

UPXPR修改了一些UPX加壳的标志，修复这些标识即可重新用UPX -d参数脱壳。光盘映像文件中提供的UPXPR_notepad.exe是被UPXPR处理过的记事本程序，用UPX -d命令脱壳提示“CantUnpackException: file is modified/hacked/protected;take care!!!”。

用LordPE打开该软件，查看区块信息。一般被UPXPR处理过的，其第一、二个块的块名不是UPX0和UPX1，而是其他字符。所以第一步就是恢复两个块名，将第一个块名改为UPX0，第二个块名改为UPX1。修改方法是：先选中块，单击鼠标右键，选择“edit section header”项目，在“Name”框中输入新的块名（如UPX0），如图13.65所示。

[image]

图13.65　查看区块表

如果用十六进制工具打开没处理过的UPX外壳，查看加壳UPX的版本号，在版本后有一UPX加壳标志“UPX!”，如图13.66所示。UPXPR会将此标志删除，导致UPX不能解压。

[image]

图13.66　查看UPX版本号

UPX 0.9x～1.2x各版本在标志UPX!后面的4个字节均为0C 09 ?? ??，更高版本是0D 09 ?? ??，可以用这个特殊字节来定位UPX标志位置，找到后再把它前面的4个字节改为UPX!（见图13.67）。

[image]

图13.67　恢复UPX加壳标志

这样，恢复UPX!、UPX0和UPX1标志后，就可用UPX -d命令对此文件脱壳了。

在0C 09 ?? ??这4个字节之后还有24个字节，修改其中的任何一个字节都会使UPX无法解压，所以如果不知道这些正确的数值就会无法恢复。

另一款工具UPXFIX_by_DiKeN可以很好地修复处理过的UPX外壳，它可以重构UPX外壳，处理后，用原版UPX -d命令脱壳。

2．手动脱**UPX**的壳

UPX壳既破坏了输入表也破坏了重定位表，尽量使用其自身命令脱壳，实在没办法了再尝试手动脱壳。

用UPX v3.01将EdrLib.dll文件加壳，用PE工具查看其PE信息。

EntryPoint: E640h

ImageBase: 400000h

再查看其区块信息，如图13.68所示。UPX加壳后已将区块重新组织，分别是UPX0、UPX1、UPX2等。其中UPX0的Raw Size是0，UPX是将解压缩后的原始文件数据映射到此区块中。UPX的解压缩执行代码在UPX1里，被压缩的原始数据放在UPX1和UPX2中。

[image]

图13.68　查看区块表

针对UPX的壳，装载后，可以不跟踪，在代码窗口里一直往下翻屏，就能发现类似下面的跳转代码，一个跨段指令转到OEP，如图13.69所示。

[image]

图13.69　跳到OEP

由于DLL重定位，此时对内存操作的指令被修改了。例如这句：

[image]

为了得到与加壳前一样的文件，必须找到重定位的代码，跳过它，让其不被重定位。重新加载DLL，对上句重定位的地址3D1267h下内存写断点，中断几下，就可来到重定位的处理代码处。

[image]

UPX壳已将原基址重定位表清零，重定位操作时，使用其自己的重定位表。地址3DE7B4h处ESI指向UPX0区块的VA，本例为3D1000h，为了让代码以默认ImageBase的值400000h重定位代码，可以在这句强制将ESI的值改为401000h。来到这句后，双击ESI寄存器，改成

401000h，然后按F4键来到3DE7C7h这里。此时代码段的数据没被重定位：

[image]

此时可以Dump了。运行LordPE抓取DLL映像，并保存为upx_dumped.dll。

运行ImportREC，手工判断DLL的IAT位置和大小并填上，最后单击“Get Imports”按钮，重建输入表后的文件保存为upx_dumped_.dll，如图13.70所示。

[image]

图13.70　ImportREC重建输入表

接下来用ReloREC工具构造一份新的重定位表，首先将UPX外壳这些要重定位的RVA提取出来。在处理重定位代码语句中，下面这句就是对代码重定位，其中EBX保存的就是要重定位的地址。

[image]

补丁的思路是找块代码空间，跳过去执行补丁代码，将重定位的地址转成RVA，并保存下来。如下语句跳到补丁代码处：

[image]

补丁代码：

[image]

3E0000h这个地址是OllyDbg的插件HideOD临时分配的，其初始值设为3E0010h，如图13.71所示。

[image]

图13.71　分配空间保存重定位的RVA

执行完补丁代码，数据窗口将保存需要重定位的RVA，执行菜单“Binary/Binary copy”功能将数据复制出来，并用WinHex将提取的数据保存为Relo.bin。然后在dumped_.dll里找一块空白代码处保存重定位表（一般在UPX1或UPX2区块里找），在这里选择C000h处。最后，参照13.5.4节，用ReloREC完成重定位表的修复。

13.8.2　ASPack外壳

ASPack外壳运行时，曾有一段时间将程序完全解密，此时内存映像是加壳前的状态，输入表、重定位表都是完整存在的，没有被破坏。正是由于ASPack保留了加壳前程序完整的状态，因此其兼容性极好。ASPack脱壳很简单，适机抓取内存映像，再修正PE头的输入表、重定位

表的地址即可。

用ASPack 2.12将EdrLib.dll文件加壳，查看其PE信息。

EntryPoint: D001h

ImageBase: 400000h

1．寻找OEP

寻找DLL的OEP有两条路可以走：一是载入时找，二是在退出时找。本节采用第二种方法。

OllyDbg加载EdrLib.dll，中断在外壳代码第一行。代码如下：

[image]

在外壳的入口点按F2键设一个断点，按F9键让EdrLib.dll运行，DLL装载成功后，关闭loaddll.exe界面，即会卸载DLL文件，将再次中断在外壳的入口点处。

[image]

注意，在“CALL 003DD00A”语句处要按F7键跟进，不然按F8键程序就会运行起来。这里的call语句不是什么真正的过程调用，无非是变形的jmp跳转语句而已。要识别它也不难，看看它跳转的地址是否就在附近。如果是就按F7键，而不要按F8键。

由于DLL已解压，因此外壳不会再次对DLL文件解压缩，3DD02Fh一行将跳过外壳解压代码直接到OEP处。跟踪EXE文件时，也可直接在DD02Fh这一行跳转来定位到OEP的处理代码处。

[image]

EdrLib.dll装载后，基地址不是默认的400000h，新的基地址是3D0000h。此时的OEP的RVA值为1240h。

2．解压分析

用LordPE查看EdrLib.dll的区块信息（见图13.72）。ASPack加壳时没合并区块，各区块的RVA仍与加壳前一样。.aspack与.adata是外壳的执行程序和数据，脱壳后可以去除。

[image]

图13.72　区块信息

ASPack外壳依次将.text、.rdata、.data、.reloc区块解压，并放到正确的位置上。当这些区块解压结束后，内存映像是加壳前的原始状态，外壳还没有来得及进行进一步处理，这个时候正是得到完整映像文件的好时机。

由于外壳会向区块里写数据，可以对区块地址设内存断点。.text区块的RVA为1000h，加上映像基址，本例结果为3D1000h。在数据窗口中，对此地址下内存写断点，同时监视3D1000h内存数据变化。中断几

次，会发现3D1000h内存数据已还原。此时代码如下：

[image]

上面代码的作用是ASPack外壳将已还原的区块数据放回将要执行的区块空间里，会循环执行数次，直到所有的块都还原后，内存中是完整的原程序，此时可将内存映像抓取下来。操作时，只需要在3DD1A9h一行设断，中断后抓取内存映像文件，保存为dumped.dll。

3．输入表

由于ASPack外壳还没来得及破坏输入表，只要找到输入表的地址即可。根据API函数的调用，确定IAT的RVA是7000h～70E4h。重新加载EdrLib.dll，在IAT里任意选一地址设置内存写断点。中断如下：

[image]

根据跳转，向上来到输入表处理的代码处：

[image]

从3DD278h这句可得知输入表RVA为7694，大小可以用十六进制工具查看dumped.dll。根据IID结构，很容易得知大小为3Ch。

为了进一步巩固输入表知识，讲述另一种方法确定输入表的地址。dumped.dll文件里输入表是完整存在的，因此以kernel32.dll为突破口，反推IID结构的地址。用十六进制工具打开dumped.dll文件，查找“KERNEL32.dll”字符串（因为一般程序输入表中肯定存在此字符）。会发现有三处，第一处是原程序的输入表（见图13.73），第二、三处是在外壳的代码里（即在.aspack块里）。

[image]

图13.73　显示KERNEL32.dll字符串

因为dumped.dll文件是内存映像，所以其RVA值与文件偏移值相等。图13.73显示的KERNEL32.dll的地址为77D0h，该地址是IID结构中Name项的值，而Name值存在形式为“D0770000”。然后以Hex Values模式查找十六进制，结果如图11.74所示。

[image]

图13.74　IID结构

上面显示的就是输入表IID数组，共有两个数组，如表13-5所示。

表13-5　十六进制工具中显示的IID数组

[image]

第一个IID数组地址就是输入表的地址，结果为7694h。

4．基址重定位表

EXE文件一般不需要重定位表，这步可略过。ASPack没破坏重定位表，因此只需要确定重定位表的地址和大小即可。

进入OEP后，寻找一句需要重定位的地址。

[image]

地址3D1267h会被重定位，重新加载DLL后，对3D1267h设置内存写断点。中断如下：

[image]

外壳程序在3DD1E5h开始模拟Windows系统重定位代码，此时ESI的值就是重定位表的起始RVA，本例为C000h。这段初始化代码以ESI为指针，取重定位表的数据，当执行结束后，来到3DD257h，此时的ESI中的值是重定位表的结束地址，本例为3DC5C0h，转成RVA为C5C0h，因此重定位表的大小为5C0h。

如果熟悉重定位表，可以用十六进制工具直接查看dumped.dll文件，以确定重定位表的地址和大小。重定位表一般以“00100000”开始，并且在十六进制工具中右边的字符栏显示的是可见的ASCII字符，因此很好辨认。

5．PE文件修正

（1）OEP修正

用PE编辑工具LordPE打开Dump.dll程序。将1240h填进EntryPoint域中，单击“Save”按钮保存。

（2）输入表修正

用LordPE打开Dump.dll，单击“Directories”按钮打开目录表，在ImportTable的RVA域中填入7694h，大小为3Ch（此值无关紧要，可填个比0大的数字），单击“Save”按钮保存修改，如图13.75所示。

[image]

图13.75　修正输入表的地址

（3）基址重定位表修正

用LordPE打开Dump.dll，单击“Directories”按钮打开目录表，在Relocation的RVA域中填写C000h，Size域中填写5C0h。

（4）删除无用区块

程序中还有两个外壳使用的块已不需要，可以删除它们。用LordPE打开dumped.dll，单击Sections打开区块表窗口，用鼠标右键菜单中的“Wipe section header”命令删除.aspack（ROffset = D000h）和.adata（ROffset = F000h）块。再用十六进制工具打开dumped.dll，删除D000h后的数据。

13.9　加密壳

此类壳以加密保护为重点，用尽了各种反跟踪技术，保护重点是在OEP隐藏和IAT加密上，甚至是虚拟机加密技术。由于看雪论坛精华集里对加密壳进行分析的文章比较丰富，故本书不再重复了。

13.9.1　ASProtect

ASProtect是一款经典的加密壳，它曾一度代表了加密壳的发展方向，其特长在于加密算法的运用，在保证强度的前提下，有着极好的兼容性和稳定性。

由于ASProtect在软件加密领域名气太大，用其保护的软件也多，从而导致许多爱好者研究其保护机制，并找出拆解的方案。因此，ASProtect脱壳资料已很多，具体请参考看雪论坛中相关帖子。VolX的Aspr2.XX_unacker脚本能支持目前ASProtect所有壳的版本，相当于一款脱壳机了。本节不是讲解如何去脱ASProtect壳，而是与大家讨论一下ASProtect的一些保护技术点。

1．Emulate standard system functions

ASProtect可以将API入口一段代码抽到外壳里，执行完毕后，从中间调用系统API，这样对API入口地址设断的传统方法将失效。来看一个样例，加壳前的API调用语句：

[image]

用ASProtect对目标程序加壳，勾选上“Emulate standard system functions”。OllyDbg加载目标文件，来到同样的API调用处，其已将DialogBoxParamA函数的开头部分代码抽到外壳空间，执行完毕后，再跳回DialogBoxParamA继续执行。

[image]

调试这类保护的程序，可以将断点设到API函数结尾返回处。

脱壳时，必须让外壳将正确的API调用写回去。先来查看一下目标文件的区块信息，如图13.76所示。

[image]

图13.76　查看区块信息

外壳的入口代码在第一个区块上，外壳初始化完毕，必定会将解压出来的原程序数据填回第一个区块。因此，用OllyDbg重新加载目标程序，在数据窗口中，对第一个区块设内存写断点，本例是401000h。

设断后，按F9键运行程序，第一次中断忽略，第二次中断如下：

[image]

取消内存断点，来到A62676h这行时，按“Alt + M”键打开内存窗

口，对第一区块下内存访问断点。中断后：

[image]

这是一段对外壳进行校验的Hash函数，往下翻屏，一直来到函数的结尾处：

[image]

走出这段校验函数，来到：

[image]

外壳校验完毕，将开始处理“Emulate standard system functions”功能，因此，按“Alt + M”键打开内存窗口，对代码段再次下内存写断点，当对代码段改写时，OllyDbg就会再次中断。

[image]

这段外壳代码，是将需要模拟的API函数头部提取出来，并将程序调用API的指令改为调用外壳指令。

修复的思路是，对这段外壳代码进行补丁，改写其原来功能。首先是在IAT里搜索获得的API地址，得到函数在IAT中的调用地址，然后将这个调用地址写回原程序里。

先在A8BA97h设个硬件断点，重新加载目标程序，会中断此处。用插件HideOD临时分配一些空间写补丁代码，在临时空间键入如下补丁代码：

[image]

可以在A8BA97h这句，键入一个转移指令jmp D000000来执行补丁，但由于这样改变了外壳代码，外壳的校验会导致程序异常出错。解决这个问题可以用OllyScript脚本来改变程序流程，保留A8BA97h这处的硬件断点，执行如下的脚本。这样每当程序中断在A8BA97h这句时，脚本将把补丁的地址D00000h写进EIP，并继续运行。

[image]

执行脚本前，在A8BB01处设一个断点，当脚本执行完毕后，就会来到这里。

[image]

补丁修复结束后，按“Alt + M”键打开内存窗口，在代码段设断，可以直接来到OEP。

2．stolen bytes

stolen bytes是指外壳将程序部分代码变形，并搬到外壳段。

ASProtect不仅能将OEP代码搬到外壳里，还能将程序中的代码搬到外壳里（需要编程使用SDK）。用ASProtect对目标程序加壳，勾选上“Protect Original EntryPoint”。OllyDbg加载目标文件，用上一节的方法，可以来到此处。

[image]

这段是Advanced Import protection保护，具体可参考看雪论坛精华集。在A8BB01h一行，按F4键跳过这段，按“Alt + M”键打开内存窗口，对代码段下内存访问断点。中断如下：

[image]

走出这段代码，来到：

[image]

在A8BF2Ah一行，在数据窗口中查看EBX指向的数据，如图13.77所示。

[image]

图13.77　查看指向stolen bytes的数据

这些数据就是外壳指向stolen bytes的数据，其值为RVA，例如其中的一个数据13A0h，加上基址，其地址为4013A0h。当外壳执行到A8BF33h这句时，查看4013A0h的代码。

[image]

ASProtect外壳为了兼容性，在原来抽掉的代码处，保留了一个转移指令，其指向stolen bytes代码。

stolen bytes代码是由一些变形代码和花指令组成的，例如一个典型变形语句：

[image]

变形后：

[image]

重建stolen bytes是很困难的，如果是OEP处的代码，可以根据编译语言的特征进行恢复，或采取补区段的方法恢复。有关stolen bytes修复，读者可以参考看雪论坛精华集的相关资料。

13.9.2　Themida的SDK分析②

Themida是一款优秀的保护壳，在放弃使用驱动反调试后，强度主要靠SDK的虚拟机技术来保证。SDK之外的脱壳过程，和其他的壳没有

太大区别。下面主要讨论SDK保护代码的修复。

光盘映像文件中提供的演示程序VMTest，使用了SDK的Encode、Clear、CodeReplace和VM宏，用Themida1910加壳，关闭Protection Options中的全部选项，虚拟机保护设置为最低，处理器类型选择CISC-2，如图13.78所示。

[image]

图13.78　范例的VM加壳设置

1．Encode与Clear宏保护代码的修复

下面是使用ENCODE START/END宏保护的原始代码。

[image]

加壳后的代码为：

[image]

401001处变成了call指令。Encode宏保护实际是SMC，即执行时会解出明文代码。ENCODE_START占18字节，在第19字节即原始代码处（401013）下硬件执行断点。

[image]

不仅解出了明文代码，在原ENCODE_END的位置（401029）还有个call，执行后会将解出的代码重新加密。只要保存解码结果，NOP掉ENCODE_START/END宏对应的36字节即可。

Clear宏与Encode相似，唯一的区别是CLEAR_END为23字节，作用为擦除解出的代码，而不是重新加密。

[image]

可以在Dump之前编写OllyDbg脚本来修复这两种宏保护的代码。

2．CodeReplace与VM宏保护代码的修复

对于Themida当前版本，这两个宏是相同的，都使用虚拟机技术。加壳时将原始机器码反汇编转换为伪码，执行时由虚拟机引擎解释执行。如果想还原代码，需要分析VM解释引擎的工作方式，编写pcode解码器。Themida目前支持4种VM处理器类型，分为CISC和RISC两类，后者提供的保护强度更高，也更复杂。

出于自我保护的目的，VM解释引擎是被混淆过的变形代码。原始代码按预先定义的模式膨胀，生成的结果被划分为若干代码块，随机置换各代码块的物理位置，再用JMP指令链接起来，如下面的例子。

[image]

这段代码逻辑上是连续的，物理上被分成了4块，用3个JMP链接起来。eax和ebx为空闲寄存器，用来生成干扰指令。以粗体显示的是有用的代码。从014F562A开始对ecx的连续变换结果为0。去掉JMP和干扰代码后为：

[image]

这实际上只等于1条指令：

[image]

如果不对变形代码进行清理，很难理解handler的真正目的。遗憾的是，笔者没有完美可靠的办法，目前的做法是根据具体的代码变形模式进行匹配压缩的，得到的结果容易出错，只能用来辅助分析。希望有人能开发出更好的方法。

下面分析范例中被CodeReplace宏保护代码的执行过程。VM宏与此类似，留给读者练习。光盘映像文件中有个简单的解码程序。被保护的原始代码为：

[image]

加壳后的代码，以JMP开始执行VM保护代码。

[image]

压栈的imm32代表了pcode数据的地址，同时被用做pcode数据解码key。下面是VM入口代码。

[image]

[image]

执行初始化代码后，检测VM是否忙，忙则等待，VM不支持多线程访问。如果VM空闲，开始取pcode解释执行。清理后的取指令代码为：

[image]

handler的最后1条指令为JMP，跳到1_FetchOpcode继续取指令/执行指令循环，直到遇到特定的opcode（如ExitVm）。

在调试代码前，先了解一下VM的基本特征。最重要的是上下文结构VMCtx，保存了原程序寄存器组及VM内部使用的变量。其成员的含义要根据handler如何使用来确定。演示程序的VMCtx结构在412C0A处。

[image]

在VMCtx结构后面就是handler地址表，共167项。这里列出的数据已替换为清理变形代码后的地址，注释中是原始的handler地址。

[image]

Themida CISC-2处理器的指令长度为1字节，每条指令可带有0/1/2/4字节的操作数。是否带有操作数，可以从相应的handler代码看出来，若有操作数，也需要解码。VMCtx结构成员及handler地址表用1个字节作为索引即可寻址，opcode编码直接从0x11开始。

进入VM后，有3个寄存器有特殊含义，在handler执行过程中保持不变：

[image]

handler的实现使用了3个寄存器eax/ecx/edx，未使用ebp。可以认为这3个寄存器是VM内部的寄存器。为避免引起混淆，解码时将这3个寄存器另外命名。

[image]

pcode与被保护的原始机器码并非是一一对应关系，一条机器指令一般需要执行几条pcode才能模仿，如果再加上pcode变形，数量更多，性能损耗相当大。另外，有少量opcode由VM内部使用（如实现pcode变形），并不用于模仿原代码。

[image]

这5行pcode代码模仿原来的一条指令“mov ds:[112ADEE4],0”。

Themida VM提供的是“平面”式的保护，即被虚拟机保护的代码，如果其中有调用其他函数（或API）的代码，被调用代码不会被纳入保护。当执行到call时会退出VM，调用完成后重新进入VM，这样原始代码对应的pcode数据被明显地分为几段。另一种情况是被保护代码中含有VM不支持的指令（如浮点指令），也会到VM外来执行。

下面是范例中被CodeReplace保护代码的pcode数据，被一个call分为两部分。pcode数据的地址为进入VM时push的imm32加delta。

[image]

第1段pcode数据地址 = 7265CFC + F925B014 = 004C0D10

第2段pcode数据地址 = 7265E67 + F925B014 = 004C0E7B

在实际情况中可能并不容易迅速看出pcode数据究竟由几段组成，但在对pcode解码的过程中最终会显示出来。

限于篇幅，有关范例中被CodeReplace保护代码的pcode分析的结果，请参考光盘映像文件中提供的文档。

13.10 静态脱壳

脱壳机编写分成两类，一类是静态脱壳，另一类是动态脱壳。静态脱壳需要完全分析出壳的引导过程及解压算法，把要脱壳的程序作为数据文件输入，然后自己实现壳的数据解压过程，修正PE结构，完成脱

壳。这样可以避免调试不慎程序跑飞，很适合病毒、木马的脱壳分析，因此一些杀毒软件会集成一些静态脱壳引擎。动态脱壳机可以用调试API或虚拟机技术来实现，其加载目标文件，可以控制外壳的运行，利用壳自身解密数据再修复PE结构，相对来说比较容易实现。

13.10.1　外壳Loader的分析

编写静态脱壳机需要弄清楚壳的Loader工作过程，本文以ASPack 1.08.0版本为例讲解静态脱壳机编写的基本过程。

1．外壳第一部分

该外壳的Loader分两部分，第一部分以非压缩的方式存在，第二部分以压缩的方式存在。外壳执行时先执行第一部分，这部分将外壳的第二部分在内存中解压缩，并初始化一些数据。用IDA打开目标实例，来到外壳的入口处。分析如下：

[image]

这段代码功能就是取当前映像的基址。接着外壳调用aPLib解压引擎，将外壳的代码第二部分解压出来。相关代码如下：

[image]

[image]

这段代码中最关键的就是解压函数10250D4 call aPLib_Decode的分析，如果不能识别何种算法，需要将其算法逆向，或将解压函数的汇编代码直接提取出来，程序里内嵌汇编调用即可。本例ASPack是调用了aPLib v0.22b压缩引擎，因此写脱壳机的时候，可以寻找相应版本的aPLib SDK参考。

外壳将第二段代码解压后，覆盖到10250FDh地址处。由于外壳第二段是加密的，在IDA中查看是乱码，必须将其解压。可以用IDC脚本来实现，也可以用OllyDbg将解密后的数据提取出来，导进IDA。后者相对来说更容易操作些，用OllyDbg加载实例，停在10250E7h这行，将ESI指向的54Ah大小的数据提取出来另存为pack_2.bin。执行IDA的菜单“File/Load file/Additional binary file”，打开pack_2.bin文件，如图13.79所示。在“Loading segment”中填上102500h，“Loading offset”中填上0xFD，执行后，pack_2.bin数据就会更新到IDA中的10250FDh地址处。

[image]

图13.79　将二进制文件导入到IDA中

2．外壳第二部分

第二部分开始的代码是外壳的真正部分，主要功能是还原原始程序，并初始化IAT，如果需要重定位，则重定位，完成后就跳到真正的

程序处执行。这部分比较重要，必须分析出外壳的一些重要数据结构，如区块的数据、输入表、入口点等信息。

[image]

上面是外壳的一些重要数据结构，这些数据在内存中的偏移是固定的，如入口点Entrypoint的偏移为11Dh。脱壳机工作时，需要从这些偏移中获取相关的数据及结构。

外壳的第二部分代码如下：

[image]

[image]

这段代码是将各区块的数据解压，并复制到内存映像指定的位置上。其中一段代码是修复代码段E8E9的功能，需要注意一下。相关汇编代码如下：

[image]

外壳为了提高压缩率，将CALL、JMP指令修正了一下。下面两句指令是压缩前的指令，都是CALL到同一地址，但其机器码不同。代码如下：

[image]

现在外壳改为用相对基址的偏移来改写指令，100832Ah的RVA为832Ah，相对基址1000h的偏移为732Ah，1007C8Eh + 732Ah = 100EFB8h，因此改写的指令如下：

[image]

改写后，CALL调用改成相对于Base的偏移，而不是当前指令的偏移，这样提高了机器码的重复率，压缩率也就提高了。

外壳运行时，按这个相反过程将原指令恢复。用高级语言来描述就是：

[image]

原始数据解压恢复后，如果需要重定位，外壳就会对代码进行重定位。由于ASPack没破坏原始重定位数据，因此只需要得到重定位表的地址就可修复了。

[image]

从上面的代码可知，重定位表的RVA外壳保存在偏移115h处。

原始数据解压恢复后，外壳就会模拟Windows系统加载器，填充IAT，由于ASPack没破坏原始输入表的结构，因此此时只需要得到输入

表的地址就可修复了。

[image]

从上面的代码可知，输入表的RVA外壳保存在偏移119h处。

现在已经分析完了Loader。用伪代码描述一下它的核心流程：

[image]

13.10.2　编写静态脱壳器

通过分析，已经知道壳的加载过程，现在来分析如何取得解压缩时的重要数据。写静态脱壳的关键就是找到正确的数据，把相关信息还原或者补回去。其中必需的步骤如下：

① 数据位置的确定；

② 算法的还原；

③ 输入表、重定位表以及资源的修复。

要想正确脱壳，首先得对壳进行识别，对根本不认识的版本进行脱壳的结果是不可预知的。因此，首先来检查程序的入口，判断是否为目标。

[image]

然后，将解压函数call aPLib_Decode中的代码逆向或直接将汇编代码提取出来。具体代码详见源码中的aPLibDePack()函数。

接着调用aPLibDePack()将各区块的数据解压恢复出来，代码如下：

[image]

由于壳没有破坏输入表、重定位表，可以直接将这些数据写回PE头，否则必须重建输入表和重定位表。

外壳压缩时，通常壳会把MAINICON和VERSION等资源提取出来放在壳外面（为了能够正常看到程序图标和版本信息）。如果需要将外壳的代码区块删除，则脱壳时必须将这些资源还原到.rsrc资源区块里或重建资源。这部分代码本节没有提供，读者参考相关资料自己加上。

最后，修正入口点，优化区块信息，将修正后的文件保存到文件，完成脱壳。

注释

① 音有ké和qiào，这里一般发音ké。

② 本节由softworm编写

第7篇　保护篇

- 第14章　软件保护技术
- 第15章　反跟踪技术
- 第16章　外壳编写基础
- 第17章　虚拟机的设计

市场上虽有大量现成的保护方案可选用，如基于软件的加密壳保护和基于硬件的加密锁保护，但这些优秀的保护方案由于太流行，造成大家对其研究的透彻，反而容易被破解。所以，有必要自己实现相关的保护方法。

第14章　软件保护技术

市场上虽有大量现成的保护方案可选用，如基于软件的加密壳保护和基于硬件的加密锁保护，但这些优秀的保护方案由于太流行，造成大家对其研究的透彻，反而容易被破解。所以，有必要自己实现相关的保护方法。

14.1　防范算法求逆

设计一套合理的注册算法，应该有必要的抗分析手段，可以有效限制解密者分析注册算法，从而阻止注册机或者破解补丁的出现。

14.1.1　基本概念

软件保护的目的是向合法用户提供完整的功能，所以软件保护必然要包括验证用户合法性的环节，而这一环节通常采用注册码验证的方式实现。

（1）用户向软件作者提交用户码U，申请注册。

（2）软件作者计算出注册码R = f(U)，返回给合法用户。

（3）用户在软件注册界面输入U和R。

（4）软件验证F(U,R)的值是否合法来判定用户的合法性。

其中一些常用术语说明如下：

- 用户码U：用于区别用户身份。
- 注册码R：用于验证用户身份。
- 注册机：把R = f(U)中的小f称为注册机，掌握了注册机就有能力针对任何用户码计算出相应的注册码。
- 验证函数：把F(U,R)中的大F称为验证函数，软件使用验证函数验证注册码的合法性，即，当且仅当R = f(U)成立时，F(U,R)取合法值。
- 算法求逆：把解密者通过验证函数F推导注册机f的过程称为算法求逆，所以验证函数F的构造非常关键。

在软件注册保护的“初级阶段”，验证函数与注册机没有本质区别，即：F(U,R) = f(U) - R。这样做很危险，验证函数自身就包含注册机，解密者只需要跟踪软件的运行，直接将软件验证函数中计算f(U)的汇编代码拷贝下来就可以当注册机用，甚至根本不需要了解f的算法。

改进的做法是先求出f的反函数f^{-1}，使：U = f^{-1}(R)，然后令F(U,R) = f^{-1}(R) - U。这样做安全了许多，软件本身不包含注册机f，解密者必须在充分了解f^{-1}算法过程的基础上才能分析推导出注册机f。可能会有读者感到疑惑：假如解密者先指定R，然后直接利用验证函数计算出U = f^{-1}(R)，不就可以使用U、R来注册了吗，何必一定要推导f呢？在实际应用中，由于U、R通常以字符串的形式给出，而验证函数通常采用数值运算，所以一般会将U、R转换成数值形式U'、R'，则注册机f及注册机的反函数f^{-1}实际上都是复合函数，验证函数由于只需要检验U'、R'的合法性，因而可以不完全等于f^{-1}。

[image]

看起来解密者通过F(U,R) = [image]([image](R)) - f_1(U)推导f^{-1}比推导f要容易，关键在于f^{-1}通常是建立在ASCII编码表之上的一个变换，所以即使推导出f^{-1}也没有多大价值。

14.1.2　堡垒战术

事实上，在通信领域人们很早就开始了身份验证的研究，并发展出了散列加密和非对称加密等优秀的密码学算法，其中的MD5算法和RSA算法非常适合在软件注册算法中运用。

MD5算法通常并不被直接用来对消息进行加密，因为不存在逆算法，所以加密之后无法解密，这样的加密是没有应用价值的，MD5算法的用途在于数字签名。例如甲和乙进行通信，甲仅仅将明文A加密为B传送给乙是不够的，因为即使密文B的加密强度再高也只能防止在传输的过程中被解密而泄露内容，却不能防止破坏者直接篡改密文B。乙收到B后解密得到A，也无法确定A所包含的内容的真实性。所以甲在发送B的同时通常会在A之后署名，然后计算C = MD5(A)，将B、C一起发送给乙。乙收到B、C后，首先解密B得到A，然后同样计算C = MD5(A)，若计算出的C与收到的C相同，则可确定A真实可靠。

同样MD5算法也不适合直接被用来做注册机，假如使用R = MD5(U)做注册机，由于MD5不存在反函数，验证函数将不得不包含注册机。但是用以下办法使用MD5算法：

（1）设注册机f为：R = f(U)。

（2）设MD5(a) = b。

（3）令验证函数F为：F(U,R) = MD5[f^{-1}(R) - U + a]。

显然F的合法值应该为b，只要U、R满足R = f(U), F一定等于b。由于MD5不可逆，解密者无法通过b获知f^{-1}(R) - U + a应该等于a，也就无法获得f^{-1}的准确表达式，更无法获得注册机f。至于f^{-1}(R) - U + a表达示中虽然含有a，但解密者无法判断a和f^{-1}的关系。例如：U = f^{-1}(R) = R*5 + 19，a = 7，则F(U,R) = MD5(R*5 + 26 - U)，a和f^{-1}融为一体，叫解密者如何分得清？

实际上利用MD5算法还有很多方法可以构造F，让解密者根本看不出F和f、f^{-1}的关系，就更谈不上求逆了。这里只是举了一个简单的例子而已，相信读者完全能够进行精彩的发挥。

当然，MD5算法也有它的缺陷，正因为MD5完全不可逆，所以a必须为常数，一旦解密者获得了一对合法的U、R，他们就可以跟踪到合法的a、b值，从而获得f^{-1}，并进一步推导出注册机f。当然，前提是他们必须先设法获得一对合法的U、R。

RSA算法很容易在软件保护中进行应用：

（1）软件作者使用：R = U^d mod n作为注册机。

（2）软件使用：U = R^e mod n作为验证函数。

解密者即使跟踪软件运行的全过程，也得不到d，无法写出注册机。基本上进行RSA算法保护的软件可以说是建立了一座坚不可摧的堡垒，但是由于RSA算法众所周知，再加上软件作者通常会采用公共函数

库来实现RSA算法，所以使用RSA算法也存在一些风险。

① RSA算法本身虽然足够坚固，但使用者往往直接采用第三方公用代码。这些代码可能含有漏洞，而世界上有大量的爱好者在研究这些代码的漏洞，一旦发现漏洞，软件的安全性就可能成为陪葬品。

② RSA算法的使用者往往并不了解算法细节，可能因错误使用RSA而在不知不觉之中遭遇非常规手段的攻击。例如，在不同的软件作品中，使用不同的e、d但使用相同的n而遭到“公共模组攻击”等。

③ RSA算法在通信领域由于密文的解密过程并不暴露给窃听者，所以其加解密过程虽然同为模幂运算，但实际实现过程往往并不一致，在解密端通常会采用“中国剩余定理”进行加速，而中国剩余定理中包含对原始数据p、q的引用。这一类函数库在软件保护中使用时要非常小心，一旦软件作者选择了错误的RSA库函数，在验证函数中使用了中国剩余定理，则会导致RSA防线如同虚设。

④ 某些函数库在生成随机素数时，采用“伪随机数产生器”，即在完全相同的初始条件下会产生完全相同的“随机数”序列。解密者如果得到该函数库，就可以根据n值推断p、q生成过程，从而攻破RSA防线。

⑤ RSA算法中存在若干由某些特殊素数构造而成的“弱密钥”，某些函数库在生成随机素数时，没有淘汰这些特殊素数，导致RSA防线在数论高手面前虚弱无力。

14.1.3　游击战术

软件保护中的游击战术就是将验证函数F分解成多个互不相同的F_i，然后将这些F_i尽可能地隐藏到程序里去。

通过任意一个F_i的验证都只是注册码合法的必要条件，而非充分条件，真正合法的注册码能够通过所有的F_i的验证。解密者找到F_i其中的任一个或任意几个，只要不能将所有的F_i一网打尽，他就无法一睹F的全貌，无法进行算法求逆。

当然，将F分解成一系列必要非充分的F_i需要较专业的数学知识，但可以使用分段函数来简单地实现这一目标。

（1）将R切分成多段R_i。

（2）构造不同的f算法，使得：$R_i = f_i(U)$。

（3）令[image]

这样做虽然有点麻烦，但绝对是值得的。例如可以让F_1使用MD5算法，F_2使用RSA算法，F_3使用自定义算法。在用户输入注册码后仅仅使用F_1进行验证，并将注册码以密文形式写入自定义格式的数据文件中，如果验证通过就显示注册成功的提示。另外两个验证函数藏起来，只有使用者执行特定的操作时才被调用，例如，在用户进行存档操作或使用某些高级功能的时候将注册码读出来再次验证。一旦任何一个验证函数发现注册码非法，就清除注册码并将软件恢复为未注册状态。

游击战第二个宗旨是虚虚实实。对于解密者来说，遇到游击战术会非常被动，除非他找到的验证函数已经能够将U、R形成一对一的对应关系，否则永远不能确定软件中是否还埋藏着其他的验证函数，而事实上软件作者根本没有必要让U、R形成一对一的对应关系，验证函数个数的不确定性的确很容易让试图制作注册机的解密者懊恼不已。

假如运用一点简单的线性代数的知识，可以将R_i的其中几个（注意只是其中几个，而不是全部）和F_i关联起来：

设：$R_a = 3U$，$R_b = 5U$，$R_c = 7U$，则：

[image]

这样解密者找到F_a、F_b、F_c中的任意一个，甚至无法求出R哪怕是小小的一段。一个更好的主意是让参与到线性方程组中的R_i的个数稍稍大于使用线性方程的验证函数的个数，软件作者手里持有线性方程的某一组特定解作为注册机，而解密者则无法了解验证函数到底有几个。

如果将一对U、R作为纵横坐标，看做平面上的一点，将注册机f看做由合法U、R连成的一段平面曲线，还可以构造多个空间曲面方程作为验证函数F，条件是f落在这些空间曲面之上，如果稍稍了解空间解析几何的知识，相信各位可以构造出无数个曲面方程作为验证函数，甚至还可以考虑使用参数方程，这样即使解密者获得了所有的F_i，也要有精深的数学水平才能求出f。

这里必须反复强调数学知识的重要性，不管是数论、代数、线性代数、几何、解析几何，还是微积分、概率论，都可以拿来作为软件保护的武器。

游击战第三个宗旨是战略转移。游击战术的致命弱点在于，每一个验证函数都必须访问注册码，而注册码的源头只有一个。解密者会跟踪程序从注册界面读入注册码的过程，并监控存放注册码的内存地址，一旦验证函数访问这一地址就会泄露行踪，这样注册码实际上成为了解密者寻找验证函数的一把钥匙，理论上解密者只要牢牢地抓着这把钥匙不放，就一定会找到所有的验证函数。应对的办法就是大规模的转移，软件必须不停地将注册码“搬家”，搬家的方法要多样化。

（1）内存拷贝，这种常规做法容易被解密者用内存监视断点识破。

（2）写入注册表或文件，然后在另一处代码中再读入到另一个内存地址，这种办法会被解密者的注册表、文件监视工具识破。

（3）一次将注册码拷贝到多个地址，让解密者无法确定哪一个地址是注册码的新家。

（4）在反复使用同一个函数搬家后，突然使用另一个前半部分代码相同而后半部分不同的函数进行搬家。

（5）将以上方法反复使用。

事实上，主动权永远掌握在程序员的手里，还可以用更多的方法来对付解密者。

14.2　抵御静态分析

静态分析是指从反汇编出来的程序清单上分析程序流程。反静态分析技术主要是从扰乱汇编代码可读性入手，如使用大量的花指令、将提示信息隐藏等。

14.2.1　花指令

在反汇编的过程中，存在着几个关键的问题，其中之一是数据与代码的区分问题。汇编指令长度、多种多样的间接跳转实现形式，反汇编

算法必须对这些情况做出恰当的处理，保证反汇编结果的正确性。

目前主要的两类反汇编算法是线性扫描算法（Linear Sweep）和递归行进算法（Recursive traversal），它们都有着广泛的应用。常见的反汇编工具所用的反汇编算法见表14-1。

表14-1　常见的反汇编器实现的技术

[image]

线性扫描算法这种方法的技术含量并不高，反汇编器只是依次逐个地将整个模块中的每一条指令都反汇编成汇编指令。没有对所反汇编的内容进行任何判断，而是将遇到的机器码都作为代码来处理。因此无法正确地将代码和数据区分开，数据也将被作为代码来进行解码，从而导致反汇编出现错误。而这种错误将影响下一条指令正确识别，会使得整个反汇编都错误。

递归行进算法按照代码可能的执行顺序来反汇编程序，对每条可能的路径都进行扫描。当解码出分支指令后，反汇编器就将把这个地址记录下来，并分别反汇编各个分支中的指令。采用这种算法可以避免将代码中的数据作为指令来解码，比较灵活。

巧妙构造代码和数据，在指令流中插入很多“数据垃圾”，干扰反汇编软件的判断，从而使得它错误地确定指令的起始位置，这类代码数据称之为“花指令”。用花指令来进行静态加密是很有效的，这会使解密者无法一眼看到全部指令，杜绝了先把程序代码列出来再慢慢分析的做法。

不同的机器指令包含的字节数并不相同，有的是单字节指令，有的是多字节指令。对于多字节指令来说，反汇编软件需要确定指令的第一个字节的起始位置，也就是操作码的位置，这样才能正确地反汇编这条指令，否则它就可能反汇编成另外一条指令了。

以下是一段汇编源程序：

[image]

对源程序进行编译，然后用W32Dasm进行反汇编，来看一下反汇编后的结果。

[image]

由于Linear Sweep式反汇编软件是逐行反汇编，代码中的垃圾数据E8h干扰了其工作，结果错误地确定了指令的起始位置，导致反汇编的一些跳转指令跳转的位置无效，就像这里40100Bh地址不再是一条指令的起始处，而是出现在指令的内部了。因此，如果反汇编一个程序时发现这样的特征，就可以断定该程序中使用了花指令。当然还有很多方法可使反汇编软件落入陷阱之中。

OllyDbg打开文件用的是Linear Sweep，分析代码功能（按“Ctrl + A”键）用的是Recursive traversal算法。用OllyDbg打开实例分析后，其生成的反汇编代码完全正确，那个OE8h字节并没有被反汇编，此类花指

令不能迷惑OllyDbg。如下所示：

[image]

在Recursive traversal算法中，一个十分重要的假设是对于任一条控制转移指令，其后继即转移的目的地址都能够确定。要迷惑这类反汇编器，只要让其难以确定跳转的目的地址即可。在这创建一个指向无效数据的跳转指令代码。代码如下：

[image]

用OllyDbg打开编译好了的实例，这次汇编代码识别出错了，其认为垃圾数据0E8h所在的地址401008是有效的，故将其当指令起始地址，结果导致后面指令识别出错。

[image]

通过前面的介绍，知道由于“无用的字节”干扰了反汇编器对指令起始位置的判断，从而导致反汇编的错误结果。如果能让反汇编正确地识别指令的起始位置，也就达到了去除花指令的目的了。例如，可把那些无用的字节都替换成单字节指令，最常见的一种替换方法是把无用的字节替换成NOP指令，即十六进制数90。

[image]

OllyDbg的一个花指令去除插件，就是利用这个原理来去除花指令的，根据收集的花指令特征码，将垃圾数据替换为NOP指令，从而使得反汇编工具正常工作。

14.2.2　SMC技术实现

编写SMC代码不是汇编语言的专利，但是使用汇编语言编写可以更充分并且方便地控制每一个细节，即便如此，通常调试一段SMC代码也比调试普通的程序要花费更多的时间和耐心。不过一旦掌握了这项技术，那么就可以设计出更加完善的加密方案。

在这先看一个实例：

[image]

或许一开始读者就注意到了EnData地址处的一段莫名奇妙的数据，它是干什么用的？稍作思考，不难想到那段数据是一段被加密了的代码。现在看一下DecryptFunc函数，就可以发现这个函数会对EnData地址处的77个字节进行异或，从而解密出加密的代码，异或完成之后，程序将从DecryptFunc函数返回，并且将执行已经解密后的EnData代码。这段代码的真实面目是：

[image]

原来这段代码的功能是调用MessageBoxA函数显示一段信息，然后把自身的代码清零，最后程序结束。

现在是否觉得SMC并不难理解。下面来认识一下SMC，SMC是英文Self-Modifying Code的缩写形式，也就是说，可以在一段代码执行之前先对它进行修改。利用SMC技术的这个特点，在设计加密方案时，可以把代码以加密形式保存在可执行文件中，然后在程序执行时再动态解密，这样可以有效地对付静态分析。因此如果要了解被加密的代码的功能，那么只有动态跟踪或者分析出解密函数的位置编写程序来解密这些代码。

对于前面的例子，利用单步跟踪可以很容易得到解密的代码，即便是对于静态分析，它的解密函数也显得过于简单。现在来思考一下怎样才能在加密中更好地利用SMC技术呢？

首先，可以在解密函数中把代码的解密与自身保护以及反单步跟踪、反断点跟踪结合起来。

其次，还可以利用SMC技术设计出多层嵌套加密的代码。如图14.1所示，在第一层代码中解密出第二层的代码，而第二层代码则解密出第三层的代码，依此类推。

[image]

图14.1　多层嵌套加密的代码示意图

另外，可以设计出一个比较复杂的解密函数。针对在最外层的解密函数，还可以把它分散在程序中的多处，使其隐蔽性更强。

经过以上的考虑，就可以初步实现SMC技术的反跟踪实例了。在介绍第二个实例之前，首先来了解一下它的实现方法，如图14.2所示。

[image]

图14.2　SMC的实现方法

在前面的介绍中，已经了解到SMC代码是以加密形式保存在文件中的，为此必须有一段代码实现加密功能，对未加密形式的SMC代码进行加密。在这个例子中，OSMC.ASM文件包含有完整的未加密形式的SMC代码，经过编译后可以得到OSMC.EXE文件，而TrSMC.ASM文件中包含了加密函数以实现对OSMC.EXE的加密。运行编译后的TrSMC.EXE，即可以对OSMC.EXE文件中的SMC代码进行加密，从而得到最终的SMC.EXE文件。

OSMC.ASM代码片段：

[image]

[image]

[image]

[image]

[image]

读者可以看到在程序中设置了一个标志块（数据A部分），设置这些数据的目的是为了TrSMC.EXE加密SMC代码的方便，利用这些数据TrSMC.EXE可以很容易地定位出SMC块的信息，当然TrSMC.EXE在加密完成之后，这些数据会被清除，以免被跟踪者利用。

在这个程序中共有三层SMC代码，从外到内分别是Block1块、Block2块和Block3块。每一块的解密方法大体相同，同时加入一些反跟踪的技术。这里主要分析一下Block1块的代码。为了防止动态跟踪，在Block1块中设计了如下的方法：

① 每次只解密出加密代码块的一个字节，利用循环的方式，解密出所有的加密代码。这样循环中的反跟踪代码会多次执行，以防止跟踪者不修改代码直接通过修改寄存器值的方法跳过反跟踪代码。

② 在循环体中，对当前解密函数的代码进行自身保护，防止被断点跟踪，也就是对解密函数的代码进行校验（代码A部分），并把校验值参与解密过程。这样一旦解密函数中的代码被修改，将无法正确地进行解密。

③ 计算校验值的同时在代码中还加入了利用SEH技术的反跟踪方法。在程序中利用了修改标志寄存器的方法产生一个单步异常，在SEH的异常处理函数中将会交换EAX和EDX寄存器的值（代码B部分），同时还清除了线程上下文（CONTEXT结构）中DRx成员的值，防止BPM一类的断点，这样如果异常处理程序不能正常执行，那么解密函数也无法正确解密。

④ 另外，在循环中也加入了另一种清除调试寄存器断点的方法（代码C部分），来防止动态跟踪。这样对按F7键进行跟踪的方法也进行了有效的防护。

在以上的例子中，尽量在设计中把使用SMC技术与其他反跟踪技术结合起来，这样才能充分发挥SMC的威力。另外，用SMC技术进行多层加密时，可能存在一定的调试难度。

14.2.3　信息隐藏

目前，大多数软件在设计时，都采用了人机对话方式。所谓人机对话，即在软件运行过程中，需要由用户选择的地方，软件即显示相应的提示信息，并等待用户按键选择。而在执行完某一段程序之后，便显示一串提示信息，以反映该段程序运行后的状态，是正常运行，还是出现错误，或者提示用户进行下一步工作的帮助信息。因此，解密者可根据这些提示信息迅速找到核心代码。为了安全，就要对这些敏感文字进行隐藏处理。

现在假设有如下的逻辑：

[image]

编译后的程序用反汇编工具W32Dasm反汇编得到如下的形式：

[image]

在对该段代码进行破解时，很容易由静态的反汇编文本中对文本“You see me!”的引用，快速定位到条件的判断位置，将条件转移条件改掉。

同样逻辑，首先将文字内容做了隐藏变化，然后在程序中使用到该文字的地方首先对文字内容进行还原。使用如下：

[image]

对一些关键数据进行了隐藏处理后，在一定程度上可以增加静态反编译的破解难度。如将“软件已经过期，请购买”等数据进行隐藏处理，就可以有效防止那些利用静态反汇编，根据这些信息快速找到程序判断点进行快速破解。信息隐藏实现思路有多种，比如将要显示的字符加密存放，需要时，解密后再显示。

14.2.4 简单的多态变形技术

病毒的世界里总是充满了多态（Polymorphic）变形（Metamorphic）和混乱（Obfucscation），从ASProtect开始，外壳中也流行这些东西了。除了虚拟化（Virtualization）外，这些技术的确拥有非常好的抗分析效果。

多态和变形（也称为变态）两个术语不是特别容易区分开，事实上也没有太多的必要区分，不过从病毒制造者的角度来看，多态引擎往往意味着它会把病毒用某种算法编码，算法可能是即时生成的，也可能是密码学算法，然后引擎会生成一个充满了干扰指令的解码器，以便在运行时对病毒代码解码，这样可能会使依靠静态扫描特征码的杀毒软件失去作用。不过，现在的杀毒软件可以在运行期扫描特征码，所以多态的效果大不如前，并且因为多态要还原出明文代码，因此对于反调试也没有多大作用。多态就像代码的一层壳，过去就没什么了。

Z0MBiE的Kewl Mutation Engine（KME552）引擎变化很复杂，用堆栈解码，静态分析还是比较困难的。不过杀毒似乎不喜欢这个引擎，遇到了直接认为是Win32 Crypt病毒。B0z0的Expressway To My Skull(ETMS)能选择算法，生成的代码看起来也很复杂，可惜如果做壳只能支持EXE，因为生成的Decryptor只能在固定的EIP运行。tElock 0.98用到了 Benny's Polymorphic Engine（BPE32），可以生成无用的SEH干扰调试，可惜它加密数据仅仅是XOR，很容易被已知明文攻击，因此只推荐作为编写polymorph的一个例子，不要应用到实际当中。

变形则是把一段代码重新编码，虽然仍然使用x86指令集，但是例如“add eax, 5”可能会被换成等价的5个inc eax，并且可以用jmp打乱代码的顺序。诸如此类的变换使代码迅速膨胀，因此注重小体积的病毒并不过多地用变形，而在壳中可以不去关心体积，Themida保护一个几KB的小文件会膨胀到将近2MB，尽量地将代码变形，可以很好地干扰分析人员。ExeCryptor有不错的强度，就在于它有一个很好的变形混乱引擎，可以把壳代码变得非常难看，当然也非常难以跟踪分析。不过，作者没有对变形引擎自身保护，forgot曾经修改了它的主程序，迫使它产生了一个没有变形过的外壳体，很轻松就分析完了它的整个过程。因此，如果在壳中使用变形引擎，不要忘记保护主程序的引擎代码。

变形引擎编写比较困难，因为它需要一个反汇编引擎，还要能对代码块进行分析，似乎没有见到适合用来保护代码的开源作品，不过Z0MBiE的Mistfall 2.0实现了类似的功能，可以作为参考。

混乱通常以多态变形引擎的垃圾生成器出现，一般是指一些花指令和无用的代码。有时候变形也被叫做混乱，并且伴随着扩散，这里所谓的扩散跟香农的信息论不是一回事，只是对混乱过的代码再次混乱，更进一步地消除原始代码的特征。

要想还原变形，就必须写一个相应的收缩程序（Shrinker，因为变形也被称为膨胀），这也同样需要一个反汇编引擎，在代码中插入的数据一直是所有反汇编引擎头疼的东西，即使是IDA也没有把握区分出谁是数据谁是代码，因此推荐在引擎中插入一些难辨真伪的数据，使反汇编掉入陷阱，这样就很难编写变形代码的清除程序了。

变形引擎主要来自病毒，VX Heaven（http://vx.netlux.org）上有非常多的病毒资料，几乎所有的病毒引擎都可以下载。

14.3 文件完整性检验

在软件保护方案中，建议增加对软件自身的完整性检查，以防止解密者修改程序以达到破解的目的。这包括对磁盘文件和内存映像的检查，DLL和EXE之间也可以互相检查完整性。

文件完整性检验的原理就是文件发布时用散列函数计算文件的散列值，并将此值放在某处，以后文件每次运行时，重新计算文件的散列值，并与原散列值比较，以判断文件是否被修改。

14.3.1 磁盘文件校验实现

由于CRC-32算法的代码比较简短，故本节的自校验实例用CRC来实现。CRC算法可以对一段字符串进行CRC-32转换，最后可以得到一个4个字节长的CRC-32值。当要实现完整性校验时，首先将需要校验的文件当做一段字符串，计算该字符串的CRC-32值，然后在文件的某个地方储存这个CRC-32值。当文件运行的时候，对文件重新进行CRC-32计算，再与原储存的CRC-32值进行比较，如果文件有改动，则CRC-32值就变化了，这样就实现了文件完整性检查的目的。

为了简单起见，本文将计算文件CRC-32值的起始地址选在PE文件头开始处，结束位置定在整个文件的末尾，如图14.3所示。

[image]

图14.3 PE文件头

储存CRC-32值的位置可以放在PE文件头前那段空间处，也可把CRC-32值写入一个单独的文件中，然后在运行的时候读取该文件中储存好的CRC-32值，进行比较。

具体实现方法是：先写一个第三方的程序“add2crc32.exe”。可以用这个程序，打开一个目标文件，然后计算出目标文件（从PE头开始到文件结束这段数据）的CRC-32值，并把这个CRC-32值写入到PE文件头前那个空白处。

编程写好需要进行保护的程序，在这个程序里面加入CRC-32的校验模块。这个校验模块的工作过程是这样的：

① 先读取自身文件从PE文件头开始的所有内容，储存在一个字符串中；对这个字符串进行CRC-32转换，这时就得到了一个“原始”文件的CRC-32值。

② 读取自身文件先前储存的CRC-32值（PE文件头前一个字段），这个值是通过“add2crc32.exe”写进去的。

③ 把步骤①和②中得到的两个CRC-32值进行比较，如果相等，说明文件没有被修改过；反之就说明文件已经被修改了。

实现代码如下：

[image]

经编译后文件为crc32.exe，再用光盘映像文件中提供的add2crc32.exe打开crc32.exe，计算文件的CRC-32值，并写进PE文件头前面一个字段处。今后crc32.exe文件一旦被修改，就会自行发现。

解密者可能会自己计算CRC-32值，重新写入文件里。解决方法是在计算CRC-32值之前，对需要进行转换的字符串做点手脚。例如对这个字符串进行移位、XOR等操作，然后在最后比较的时候，也用同样的方法反计算出CRC-32值。或对计算出来的CRC-32值可逆变换处理一下，再写进文件里。

14.3.2　校验和（Checksum）

PE的可选映像头（IMAGE_OPTIONAL_HEADER）里面，有一个Checksum字段，是该文件的校验和。一般EXE文件可以是0，但一些重要的和系统DLL及驱动文件必须有一个校验和。

Windows提供了一个API函数MapFileAndCheckSumA测试文件的Checksum，它位于IMAGEHLP.DLL链接库里。其原型如下：

[image]

程序一旦运行后，new_checksum地址处将放当前文件的校验和，old_checksum地址指向PE文件的checksum字段。

此保护很脆弱，攻击者很容易用相关工具修正PE文件的校验和。一个比较好的办法，是将正确的校验和放在别处（如注册表里），需要时，用这个值与new_checksum比较。

14.3.3　内存映像校验

磁盘文件完整性校验可以抵抗解密者直接修改磁盘文件，但对于内存补丁却没效果，因此必须对内存关键代码数据也实行校验。

1．对整个代码数据校验

每个程序至少有一个代码区块和数据区块。数据区块属性可读写，程序运行时，全局变量通常会放在这里，这些变量数据会动态变化，因此校验这部分是没有意义的。而代码区块属性只读，存放的是程序代

码，在程序运行过程中数据是不会变化的，因此用这部分进行内存校验是可行的。

具体实现的思路如下：

① 从内存映像得到PE相关数据，如代码区块的RVA值和内存大小等。

② 根据得到的代码区块的RVA值和内存大小，计算其内存数据的CRC-32值。

③ 读取自身文件先前储存的CRC-32值（PE文件头前一个字段），这个值是通过光盘映像文件中提供的add2memcrc32.exe写进去的。

④ 比较两个CRC-32值。

这样就实现了内存映像的代码区块校验，只要内存数据被修改，都能被发现。这个方法还能有效地抵抗调试器的普通断点，因为调试器一般通过给应用程序代码硬加INT 3指令（机器码CCh）来实现中断，这样就改变了代码区块的数据，计算CRC-32值就会与原来的不同。当然用硬件断点不会影响校验值，因为其用了DR3 ~ DR0寄存器，没改变原程序代码数据。详细实现代码见本书光盘映像文件。

[image]

第10章已讲过，PE文件在磁盘中的数据结构布局和内存中的数据结构布局是一样的，代码区块在磁盘中的数据与内存映像数据是相同的。add2memcrc32.exe就是根据这个原理计算磁盘文件的代码区块CRC-32值，并写入目标文件里的。

如果程序不加壳这样就可直接发行了，但如果用加壳程序来进一步保护时，可能会出错。因为刚才是直接从磁盘文件中读取代码区块的RVA值和大小，加壳后，程序读取的是外壳的代码区块RVA值和大小，这样计算出来的CRC-32校验值当然就不对了。解决办法是编程时直接用代码区块的RVA具体值参与计算，这些具体的值可以用PE工具（如LordPE）查看。由图14.4可知，代码区块（.text）的RVA值为1000h，大小为36AEh，将这些值填进源程序中再编译即可。

[image]

图14.4　查看区块信息

虽然源程序一样，但在不同系统编译，代码区块的大小可能会不同，以当时编译的具体值为准。为了方便加壳，改进后的代码如下：

[image]

2．校验内存代码片段

在实际过程中，有时只需对一小段代码进行内存校验，以防止调试工具的INT 3断点。实现代码如下：

[image]

上述代码中CRC32()函数的返回值可通过调试器跟踪得到，再填进源代码里重新编译即可。具体的汇编代码如下：

[image]

跟踪调试时，如对401014h～40102Bh之间代码设INT 3断点时，CRC校验将发生变化，从而发现程序被跟踪。实际操作时，可以不提示断点被发现，而是悄悄退出，使得校验更隐蔽。

14.4　代码与数据结合技术

一般程序的代码与数据是分开的，本文提出了一个将序列号与程序代码结合的防护方案设想，在未知正确注册码的情况下，很难被破解。其实现原理是将特征数据（如序列号）与程序的某些关键代码或数据联系起来，比如用序列号或其散列值对程序的关键代码或数据进行解密（当然这些关键代码或数据事先是在软件作者那里进行加密后才发行的）。这样，即使解密者通过修改判断跳转指令可以得到一个看似注册的版本，但是不正确的注册码只会使得解密出来的那段代码或数据全是垃圾，根本无法使用。具体说来可采用如下的方法。

① 在软件程序中有一段加密过的密文C，这个C既可以是注册版本中的一段关键代码，也可以是使用注册版程序的某个功能所必需的数据。

② 当用户输入用户名和序列号之后计算解密用的密钥：密钥＝F（用户名，序列号）。

③ 对密文进行解密：明文M＝Decrypt（密文C，密钥）。

④ 对解密出来的代码加上异常处理代码，如序列号不正确，产生的垃圾代码定会导致异常发生：如序列号正确，则生成代码正常，不会导致异常产生。这一步也可采用第⑤步。

⑤ 利用某种散列算法计算解出来的明文M的校验值：校验值＝Hash（明文M）。散列算法可以采用MD5、SHA等。检查校验值是否正确，如果校验值不正确，说明序列号不正确，就拒绝执行。

14.4.1　准备工作

先选择一段代码作为要加密的数据，可以选择软件某个功能的核心代码。本文将用一实例Codedata.exe来演示，此例选择菜单“File/Open”功能代码。为了编程时能方便处理这段代码，用begindecrypt和enddecrypt两个标签将关键数据括住。代码如下：

[image]

编译后begindecrypt和enddecrypt之间的数据称为明文M。程序编译好后必须对明文M加密处理（可以用光盘映像文件中提供的Encrypter.exe工具进行处理），得到的数据就是密文C，这样才能发行。

软件执行时，调用软件自定义某种算法计算解密用的密钥：

密钥k = F（用户名，序列号）

然后对密文C进行解密：

明文D = Decrypt（密文C，密钥）

再利用SEH来处理加密代码，如解密出的是乱码，则会触发异常，这样就可利用SEH告知用户注册失败的提示消息。

源码如下：

[image]

[image]

14.4.2　加密算法选用

解密算法应该是整个设计的关键。确保算法应该无法让人逆推，因为程序代码并不是真正的随机数据，比如某些指令出现的几率较高，因此存在着被攻击的可能性。所以Decrypt算法使用不对称算法很重要，否则攻击者可以假设明文，反推密钥进行攻击。

此例便于讲解方便，故选用了最简单的XOR来加密数据。

[image]注意：由于程序代码的数据并不是真正随机数，攻击此类XOR加密只是几分钟的事，实际应用时请选用合适的密码学算法。

具体代码如下：

[image]

14.4.3　手动加密代码

Codedata.exe程序编译好后，必须进一步处理明文M这段代码。为了能在十六进制工具中方便找到明文M这段代码，待加密的源代码用了下面两行汇编代码括住：

[image]

这样，上面的代码变成了：

[image]

文件编译好后，用十六进制工具打开文件，搜索十六进制数据“4048”，就能找到待加密的代码明文M。其中开始地址address1值为143Bh，结束地址address2值为14Feh，如图14.5黑影部分所示。

[image]

图14.5　待加密的代码块

接着必须编写一个工具Encrypter.exe（程序及源码见光盘映像文件），以执行如下函数功能：

密文C = Encypt（明文M，密钥）

然后运行Encrypter.exe工具打开待加密的文件，填写通过十六进制工具得到的address1与ddress2的值（十六进制格式），再填上所需要的注册码，此处假设为“pediy”，如图14.6所示。

[image]

图14.6　Encrypter.exe工具运行界面

单击“Encrypt!”按钮，即可将明文M加密成密文C。

14.4.4　使.text区块可写

在Win32平台上，文件编译后，.text区块的属性是只读的。但是，本节实例会向.text区块写入新的数据（“xor byte ptr[esi],al”这句指令），由于.text区块只读，这样会导致程序崩溃。

因此必须用PE工具，如LordPE或Prodump等改变.text区块的属性（characterics）为E0000020h，表示可读、可写、可执行，如图14.7所示。

[image]

图14.7　改写代码区块的属性

还有一种方法不用手工修改区块属性，而是编程时用VirtualProtect等函数修改内存的读写属性，这样就可直接向.text等区块写数据了。本例采用的代码如下：

[image]

由于此例为EXE，故没有提到重定位问题的解决。对于DLL文件，把代码作为加密对象而言，如果代码中有重定位数据，则加密后的密文解密后还需要对其进行重定位，否则这段代码就算是正确解密也无法运行。希望读者注意这点。

14.5　软件保护的若干忠告

本节将给出关于软件保护的一般性建议，这些都是无数人的经验总结。程序员在设计自己的保护方式时最好遵守这里给出的准则，这样会提高软件的保护强度。

（1）尽量开发自己的保护机制，不要过分依赖不是自己开发的任何代码。在不影响效率的情况下，保护的核心代码用虚拟机保护软件处理一下，如VMProtect等。

（2）不要太依赖壳的保护，加密壳都能被解开或脱壳，现在许多壳转向虚拟机加密方向，多利用这方面的功能。如果时间允许且有相应的技术能力，可以设计自己的加壳/压缩方法。如果采用现成的加壳工具，最好不要选择流行的工具，保护强度与流行性成反比，因为越是流行的工具越是可能已被广泛深入地研究过，即有了通用的脱壳/解密办

法。

（3）增加对软件自身的完整性检查。这包括对磁盘文件和内存映像的检查，以防止有人未经允许就修改程序以达到破解的目的。DLL和EXE之间可以互相检查完整性。

（4）不要采用一目了然的名字来命名函数和文件，比如IsLicensedVersion()、key.dat等。所有与软件加密相关的字符串都不能以明文形式直接存放在可执行文件中，这些字符串最好是动态生成的。

（5）尽可能少地给用户提示信息，因为这些蛛丝马迹都可能导致解密者直接深入到加密的核心。比如，当检测到破解企图之后，不要立即给用户提示信息，而是在系统的某个地方做一个记号，随机地过一段时间后使软件停止工作，或者装作正常工作但实际上却在所处理的数据中加入了一些垃圾。

（6）将注册码和安装时间记录在多个不同的地方。

（7）检查注册信息和时间的代码越分散越好。不要调用同一个函数或判断同一个全局标志，因为这样做，只要修改了一个地方则全部就都被破解了。

（8）不要依赖GetLocalTime()和GetSystemTime()这种众所周知的函数获取系统时间，可以通过读取关键的系统文件的修改时间来得到系统时间的信息。

（9）如果有可能，可以采用联网检查注册码的方法，并且数据在网上传输时要加密。

（10）编程时在软件中嵌入反跟踪的代码，以增加安全性。

（11）在检查注册信息的时候插入大量无用的运算以误导解密者，并在查出错误的注册信息之后加入延时。

（12）给软件保护加入一定的随机性，比如除了启动时检查注册码之外，还可以在软件运行的某个时刻随机地检查注册码。随机值还可以很好地防止那些模拟工具，如软件狗模拟程序。

（13）如果采用注册码的保护方式，最好是一机一码，即注册码与机器特征相关。这样一台机器上的注册码就无法在另外一台机器上使用，可以防止有人散播注册码；并且机器号的算法不要太迷信硬盘序列号，因为用相关工具可以修改其值。

（14）如果试用版与正式版是分开的两个版本，且试用版的软件没有某项功能，则不要仅仅使相关的菜单变灰，而是彻底删除相关的代码，使得编译后的程序中根本没有相关的功能代码。

（15）如果软件中包含驱动程序，则最好将保护判断加在驱动程序中。因为驱动程序在访问系统资源时受到的限制比普通应用程序少得多，这也给了软件设计者发挥的余地。

（16）如果采用keyfile的保护方式，则keyfile的尺寸不能太小，可将其结构设计得比较复杂，在程序中不同的地方对keyfile的不同部分进行复杂的运算和检查。

（17）自己设计的检查注册信息的算法不能过于简单，最好采用比较成熟的密码学算法。可以在网上找到大量的源码。

对于加密方案的设计读者应该多从解密的角度考虑，这样才可能比较合理地运用各种技术。当然任何加密方案都很难做到完美，因此，在

设计时要注意其他方面的平衡考虑。加密方案的好坏或许用IT界一句名言来描述很合适：“一个木桶能够装多少水，是由最短的那块木板决定的”。

第15章　反跟踪技术[1]

好的软件保护都要与反跟踪技术结合在一起。如果没有反跟踪技术，软件等于直接裸露在解密者的面前。这里所说的反跟踪是泛指，包括防调试器、防监视工具等内容。本章将讨论一些常用的反跟踪方法，读者可以根据实际情况在自己的软件中采用相关的技术和代码。

15.1　由BeingDebugged引发的蝴蝶效应

一个坏的微小的机制，如果不加以及时地引导、调节，会给社会带来非常大的危害，戏称为“龙卷风”或“风暴”；一个好的微小的机制，只要正确指引，经过一段时间的努力，将会产生轰动效应，或称为“革命”。

15.1.1　BeingDebugged

Win32 API为程序提供了IsDebuggerPresent判断自己是否处于调试状态，懒惰的程序员总是用它。来看一下实现代码：

[image]

这个函数读取了当前进程PEB中的BeingDebugged标志，每个运行中的进程拥有一个名为PEB（Process Environment Block，进程环境块）的结构，对它有少许了解会有助于理解后面的内容。PEB结构的内容：

[image]

[image]

接下来的问题当然是如何找到PEB地址。它储存在另一个名为线程环境块（Thread Environment Block，TEB）的结构之内。

Windows在调入进程，创建线程时，操作系统均会为每个线程分配TEB，而且FS段寄存器总是被设置成使得地址FS:0指向当前线程的TEB数据（单CPU机器在任何时刻系统中只有一个线程在执行），这就为存取TEB数据提供了途径，如图15.1所示。

[image]

图15.1　执行线程块的结构

再来了解一下TEB的结构，请注意+30h处的偏移字段。

[image]

[image]

[image]

[image]

00h处的TIB（Thread Information Block，线程信息块）结构为：

[image]

每个进程都有自己的PEB，Windows一般通过TEB间接得到PEB的地址。即通过以下语句获得：

[image]

TIB＋18h处为Self，是TIB的反身指针，指向PEB首地址，因此可以省略而直接使用fs:[30h]得到自己进程的PEB。

为了免去繁冗的定义，这里给出一个内联汇编代码的简化版IsDebuggerPresent：

[image]

根据这个原理，OllyDbg可以用插件清除BeingDebugged以隐藏调试器。

虽然在Windows 2000/NT系统中PEB本身在大多数情况下被映射到7FFDF000h处，不过值得注意的是，从Windows XP SP2后系统引入了一个特性：PEB地址随机化。每个进程的PEB地址不固定，大概有14种可能。

系统创建进程时设置PEB的地址，调用NtCreateProcess/NtCreateProcessEx，依次转向PspCreateProcess/MmCreatePeb/MiCreatePebOrTeb，在MiCreatePebOrTeb函数中根据当前时间计算随机值：

[image]

所以不能认为PEB就在7FFDF000h，不同的进程PEB地址会不一样。当然也就不能用本进程FS:[18h]的指针去读写其他进程的内容。正确的方法是使用下面的函数，取得某个线程段选择子的线性地址：

[image]

如果喜欢Native API，也可以通过NtQueryInformationProcess获得PEB：

[image]

这里采用GetThreadSelectorEntry, 下面这段代码就可以清除BeingDebugged标记了：

[image]

现在读者一定认为这个标志太愚蠢了，事实上它比你想象的要复杂一些。BeingDebugged虽然被消灭了，但是问题并不是这么简单。

15.1.2 NtGlobalFlag

先查看一下Windows 2000中的源码，BeingDebugged被清除之前发生了什么事情。相关代码如下：

[image]

[image]

可以看到如果BeingDebugged被设为TRUE，NtGlobalFlag中也会因此被设置这些标志：

[image]

回顾一下PEB的结构，+68h处就是NtGlobalFlag，用WinHex比较内存可以发现，被调试时程序的NtGlobalFlag = 70h，正常情况下却不是。因此得到一个改进的IsDebuggerPresent函数：

[image]

ExeCryptor比较早用到NtGlobalFlag做检测，刚开始让许多人莫名其妙了一段时间。注意源码中那个LdrQueryImageFileExecutionOptions函数如果成功的话，就不会改写NtGlobalFlag了，这个函数事实上读取注册表了。注册表内容：

[image]

如果在这里建一个名为进程名、值为空的子键，那么NtGlobalFlag（及其引发的）的检测都变得无效了。

现在深呼吸一下，逐渐清醒了，这个所谓的新发现也不过是一个标志而已，还是可以像BeingDebugged一样被清除掉，可惜就如引发它的BeingDebugged一样，虽然被毁灭，它留下的痕迹仍然存在。

15.1.3 Heap Magic

为了掌握NtGlobalFlag所留下的痕迹，在WRK中找找线索。相关代码如下：

[image]

[image]

[image]

其中用到了DEBUG_HEAP宏：

[image]

NtGlobalFlag因为BeingDebugged为TRUE的缘故设置了FLG_HEAP_VALIDATE_ PARAMETERS，因此RtlCreateHeap选择用RtlDebugCreateHeap创建调试堆。再去翻一翻RtlDebugCreateHeap函数中的片段：

[image]

原来还是调用RtlCreateHeap，看来关键起作用的标记是：

[image]

回到RtlCreateHeap搜索这些文字，第一个标记看起来是为了防止从RtlCreateHeap到RtlDebugCreateHeap又回到RtlCreateHeap而做过多的重复工作。不过后面两个周围却有一些有趣的内容：

[image]

没想到调试堆里面还会被填充一些奇怪的东西，相关的定义也很容易找到：

[image]

既然堆里面被填充了东西，就可以编写以下代码测试一下：

[image]

然后用OllyDbg看看返回的指针里面都是什么：

[image]

可以看到果然有很多的BAADF00D和FEEEFEEE，还有ABABABAB。参考前面的内容，在Image File Execution Options中创建一个键，看看正常情况下的堆内容：

[image]

只是一堆没有初始化的数据而已，没有那些特殊的Magic标记。写一个程序，在堆中搜索那些奇怪的标记，如果出现了很多的话（比如10次以上），那么程序就被调试了：

[image]

这里用到了一个小伎俩，为了尽量完整地对堆进行扫描，此处用死循环一直向下搜索内存，直到内存地址已经无效时，SEH会捕获错误，此时返回统计的结果。那些奇怪的数字习惯上被称为Magic，所以这个检测技巧也被称为HeapMagic。

除了自己从堆申请内存，ap0x在他的站点公布了一段代码，从被调试程序PEB的LDR_MODULE中也能搜索到那些用来填坑的标记，事实上，Themida中发现了一字不差的检测代码。

[image]

[image]

抛开那些BAADF00D，原先发现的Flags其实还有利用价值，往下翻翻，RtlCreateHeap下面还有一些不起眼的操作。代码如下：

[image]

这里的Flags在前面已经被NtGlobalFlag影响了，看来进程堆的Flags和ForceFlags也不会幸免，它们也会感染上那些标记。

事实上，正常情况下系统为进程创建第一个堆时，会将它的Flags和ForceFlags分别设为2（HEAP_GROWABLE）和0，在调试状态下，这两个标志通常被设成50000062h（取决于NtGlobalFlag）和40000060h。下面提供的是HEAP结构，一会儿就会需要它。

[image]

[image]

请再次回顾PEB结构，并且注意+18h处的ProcessHeap，又写出一个似乎很隐蔽的标记检测代码：

[image]

15.1.4 从源头消灭BeingDebugged

首先系统创建进程的时候设置BeingDebugged = TRUE，后来NtGlobalFlag根据这个标记设置FLG_HEAP_VALIDATE_PARAMETERS等标记。在为进程创建堆时，又由于NtGlobalFlag的作用，堆的Flags被设置了一些标记，这个Flags随即被填充到ProcessHeap的Flags和ForceFlags当中，同时堆的内存也因而被填充了很多BAADF00D之类的东西。这样调试器就会被检测到。

分析这个流程会发现，一切祸根只不过是很久以前系统设置了一个BeingDebugged。犹如某地上空一只小小的蝴蝶扇动翅膀而扰动了空气，长时间后可能导致遥远的彼地发生一场暴风。因此要从源头制止这一切，在后面的事情没有发生之前改写这个值，那么所有的历史都会改写了。

系统确实给了这个时机，在编写调试器（或调试器插件）时，创建进程并调用WaitForDebugEvent后，在第一次LOAD_DLL_DEBUG_EVENT发生的时候置BeingDebugged = FALSE就可以了。但是会发现这样没法中断在系统断点了，所以在第二次LOAD_DLL_DEBUG_EVENT的时候要将BeingDebugged置为TRUE，之后

会停在系统断点，此时可以安全地清除BeingDebugged了。

笔者画了一个小表格来总结这一段代码，代码如下：

[image]

至此，一次性地解决了BeingDebugged、NtGlobalFlag、HeapFlags、HeapForceFlags和HeapMagic，这种感觉真好。

15.2　回归Native：用户态的梦魇

在Windows这个隐藏了许多秘密的系统里，该相信调用的函数吗？调用的函数还是原来的那个吗？或者确定到底调用了谁？

15.2.1　CheckRemoteDebuggerPresent

打开MSDN，除了IsDebuggerPresent之外，还有一个检测调试器的函数（本节不作特别说明的，都是指用DebugAPI实现的用户态调试器）。

[image]

看上去用法很简单，现在写一个测试程序看看效果。由于笔者的SDK版本太旧，这里用GetProcAddress获取函数地址后再调用。代码如下：

[image]

分别在调试器下和正常情况下测试，判断结果都很准确，令人满意。读者可能怀疑它是否也是读取BeingDebugged判断调试器？那就清除掉这个标记试试。

用OllyDbg加载程序，如果有IsDebuggerPresent插件，可以直接使用它来解决。也可以手工来解决这个问题，在OllyDbg的数据窗口按“Ctrl + G”键，输入：fs:[30]＋2，或在命令行里输入。若是对30和2这些偏移量仍感迷惑的话，请复习一下TIB和PEB结构，按回车键可以看到BeingDebugged的值，如图15.2所示。

[image]

图15.2　查看BeingDebugged

7FFDE002h处的01就是BeingDebugged，它的值为TRUE，在01上单击后按“Ctrl + E”键，把它修改为00，结果就是IsDebuggerPresent这个函数检测不了调试器了。测试后会发现CheckRemoteDebuggerPresent给出的判断一如平常，这不禁令人好奇它内部的构造了。

15.2.2　ProcessDebugPort

在OllyDbg的CPU窗口按“Ctrl + G”键，输入

CheckRemoteDebuggerPresent后按回车键，查看这个函数的汇编代码。代码如下：

[image]

[image]

如果感觉汇编代码不太直观，把它翻译成C语言，是这个样子：

[image]

抛开那些不需要过分关心的错误检查语句，会发现起作用的只有一行代码，就是对NtQueryInformationProcess的调用。NtQueryInformationProcess不是Win32 API，而是Native API。至于Native API又是什么？很快就会了解到，但现在可以暂且认为它们只是一些普通的API函数，不要为此伤脑筋。

大多Native API是Microsoft尚未文档化的（Undocumented），但Gary Nebbett写了一本非常酷的参考手册《Windows NT 2000 Native API Reference》，一切可以从书中找到答案。先看看这个函数的原型：

[image]

读者大概会发现函数开头变成Zw了，而不是Nt了，事实上在用户态中它们是同一个函数的两个名字。ZwQueryInformationProcess根据不同的ProcessInformationClass查询关于一个进程对象的信息，上面代码显示CheckRemoteDebuggerPresent查询了7号信息。可以从下面这个列表中找到它的意义：

[image]

[image]

数字7代表ProcessDebugPort，具体含义为：

[image]

终于了解到，CheckRemoteDebuggerPresent实际上调用了NtQueryInformationProcess，查询了某个进程的ProcessDebugPort，这个值是系统用来与调试器通信的端口句柄。NtCurrentPeb()- > BeingDebugged可以被随意地清除掉而不影响调试，但若将调试端口设置为0，系统就不会向用户态调试器发送调试事件通知，调试器当然就无法正常工作了。

如果注意到ProcessInformationClass列表中，ProcessDebugPort既支持Query也支持Set操作，自然会联想到通过与NtQueryInformationProcess相应的NtSetInformationProcess函数将DebugPort设为0，导致调试器无法与被调试进程通信来使之失效，这是个好想法。不过请注意看上面的一段关于ProcessDebugPort的说

明，“The debug port can be set only if it was previously zero”，由于程序被调试时DebugPort已经被系统设为非零值，之后就无法再设置了，因此这个想法不能转化为现实了。

别灰心，我们已经想到破坏调试器与被调试程序之间的通信，只是具体实施上遇到了一些困难。

15.2.3　ThreadHideFromDebugger

既然NtSetInformationProcess这条路走不通，就换个思路。翻开《Windows NT 2000 Native API Reference》，搜索Debugger一词，会发现一些新的线索，如ZwSetInformationThread：

[image]

这个函数可以设置一个线程相关的信息，看看ThreadInformationClass列表能发现什么？

[image]

最后一行的ThreadHideFromDebugger非常显眼：

[image]

为线程设置ThreadHideFromDebugger可以禁止某一个线程产生调试事件，这里有一个小程序，调试它看看到底产生了怎样的效果。

[image]

编译好之后用OllyDbg打开运行，会发现OllyDbg中什么也看不到了，如图15.3所示。

[image]

图15.3　OllyDbg打开后的结果

这是由于程序已经退出了，调试器打开的进程句柄不再有效。OllyDbg没有收到退出通知，或者说系统根本就不通知OllyDbg，它仍在试图反汇编进程的内存数据，可惜都是徒劳。

这个ThreadHideFromDebugger是不是把调试端口设置为0了？在Windows 2000的代码中可以找到ZwSetInformationThread是如何对ThreadHideFromDebugger进行处理的。代码如下：

[image]

可以看到，这里只是将Thread对象的HideFromDebugger成员设置为TRUE了，那么再搜索一下引用了HideFromDebugger的代码，有很多处，随便选一个看看。比如：

[image]

[image]

注释里说，如果当前进程成功映射了一个映像，就会调用这个函数。如果进程跟一个调试端口关联起来，就会通知调试器发生了LOAD_DLL_DEBUG_EVENT事件。若线程的HideFromDebugger为TRUE，代码在中间就已返回，调试器对发生的事情就一无所知了。

SoBeIt在《Windows异常处理流程》中也提到过这个HideFromDebugger，部分内容如下：

[image]

ThreadHideFromDebugger与直接将DebugPort清零的方法异曲同工，是个效果不错的反调试技巧。一会儿将会看到，它的价值不只于此。

15.2.4　Debug Object

一直以来，都紧盯着被调试的进程不放，现在来看看调试器。如果想要了解一点Windows调试器的一些内幕，推荐Alex Ionescu的系列文章《Windows User Mode Debugging Internals》、《Windows Native Debugging Internals》以及《Kernel User-Mode Debugging Support(Dbgk)》，它们包含了Windows调试机制所有的谜底，在OpenRCE.org上可以在线阅读。

尽管用OllyDbg看汇编代码也不是很复杂，不过大部分代码都可以从ReactOS项目中找到相应的、模仿得非常逼真的C语言源代码，笔者打算直接引用它们，可以让眼睛休息一下，不必为大量的push、call指令烦恼。

调试器与被调试程序建立关系有两种途径：在创建进程时设置DEBUG_PROCESS，或者调用DebugActiveProcess附加到某个已运行的进程上。建立新进程时有太多与调试无关的操作，因此以后者为入手点研究。代码如下：

[image]

[image]

Windows的很多函数都是价值不高的包装函数（Wrapper），一层层地向下调用，所以只能一层层地向下看。DbgUiConnectToDbg函数：

[image]

显然调试器创建了一个DebugObject，并且存储在了NtCurrentTeb()- > DbgSs Reserved[1]。这个域虽然名字古怪，事实上它就是调试器用来保存DebugObject句柄的地方，普通的进程中DbgSsReserved[1]应该为NULL，反过来若不为NULL，则它是一个用户态的调试器的进程。

Ntdll中有导出函数可以操作这个域：

[image]

之所以把这两个函数拿出来，是因为如果想针对DebugObject来判断某个进程是不是一个调试器，对TEB中的DbgSsReserved[1]偏移量硬编码也许不是一个特别好的方案，但可以从DbgUiGetThreadDebugObject函数体中得到准确的偏移量：

[image]

再回过头来看DebugObject是如何被创建的：

[image]

[image]

[image]

ZwCreateDebugObject事实上调用了ObCreateObject创建对象，因此可以用ZwQueryObject查询所有对象的类型，若发现名为“DebugObject”的数目不为0，那么系统中就存在有调试器。

[image]

ObjectInformationClass取值见下面列表：

[image]

设置ObjectAllTypesInformation就可以获得全部对象类型的信息，结构如下：

[image]

OBJECT TYPE INFORMATION：

[image]

注意ObjectAllTypesInformation的Remark“The ObjectHandle parameter need not contain a valid handle to query this information class”。因此需要将ObjectHandle设为NULL，Exetools上Peter[Pan]写了这一检测的完整实现，详细代码见光盘映像文件。恢复那些被注释掉的行，在有无调试器的环境下运行，可以清楚地看出对象类型结构在不同情况下的区别。

无调试器时：

[image]

有调试器时：

[image]

不过如此检测只能说明系统中存在一个调试器，却不能确定这个调试器正在调试当前的程序。如果想给Cracker一点惩戒，却无心伤害了普通用户就不合适了。这有点“宁可错杀一千，不可放过一人”的意味，过于苛刻。

不过，在实际应用中，正常用户运行一个3D游戏还开着调试器确实不太合理。检测到DebugObject就去重新启动系统的做法自然不可取，给予提示要求用户关闭调试器的方式亦容易暴露出弱点，默默地退出是比较好的做法。

15.2.5 SystemKernelDebuggerInformation

前面在《Windows NT 2000 Native API Reference》中搜索关于debugger的篇章，事实上忽略了一个函数ZwQuerySystemInformation：

[image]

当SystemInformation = SystemKernelDebuggerInformation的时候可以判断是否有系统调试器存在：

[image]

依循惯例，下面还是一个实例：

[image]

跟ZwQueryProcessInformation没有多大区别，不过请看SystemKernelDebuggerInformation的说明，其中提到这个功能检测的是“kernel debugger”，直译就是系统调试器，SoftICE是系统调试器，但这里的“kernel debugger”却不是SoftICE这样的调试器。这里有必要说一些调试器之间的差异了。

硬件调试器不在本文的讨论范畴内，在软件实现的调试器中，最明显的分界线莫过于用户级调试器和系统调试器了。前面已经提到过用户级调试器，例如VC和OllyDbg是使用DebugAPI来开发的，自身也只是一个Ring 3级应用程序，故而能做的事情有限，只能调试应用程序，无法中断内核，自然也就无法调试驱动程序。

而系统调试器则拥有更大的权利，比较流行的系统调试器就是SoftICE、Syser Debugger等，这一类调试器实现方法比较底层，调试起来速度也比较快。

比较奇特的就是WinDbg了，WinDbg可以用于Kernel模式调试，也可以进行用户模式调试，还可以调试Dump文件。推荐SoBeIt的《Windows内核调试器原理浅析》，对WinDbg和SoftICE的实现原理有比较详细的分析，文中提到WinDbg双机调试时会以Debug方式启动系统。摘自部分内容：“内核调试器在一台机器上启动，通过串口调试另一个相联系的以Debug方式启动的系统，这个系统可以是虚拟机上的系统，也可以是另一台机器上的系统（这只是微软推荐和实现的方法，其实像SoftICE这类内核调试器可以实现单机调试）。很多人认为主要功能

都是在WinDbg里实现的，事实上并不是那么一回事，Windows已经把内核调试的机制集成进了内核，WinDbg、kd之类的内核调试器要做的仅仅是通过串行发送特定格式数据包来进行联系，比如中断系统、下断点、显示内存数据等。然后把收到的数据包经过WinDbg处理显示出来。”

事实上，也只有以Debug方式启动系统时，系统中才会留下特殊的标记，上面的例子代码才能检测到“kernel debugger”的存在，如果只用WinDbg的lkd来观察内核，是无法发现的。至于SoftICE，虽然它也是系统级调试器，却不需要太多操作系统的支持，因此SystemKernelDebuggerInformation是无法探测到它的。

15.2.6　Native API

了解Native API能对Windows的内部机制有个很好的认识。值得注意的是，这些内容都是针对NT内核系统的，对Windows 9x来说不适用了。

1．认识Native API

什么是Native API？《Слежение за вызовом функций Native API》（《跟踪Native API函数调用》）已经解释得非常清楚，笔者还是要重提一下，先看图15.4。

[image]

图15.4　API的调用过程

Windows NT不但能运行Win32程序，旧的Win16、MSDOS、OS/2等系统下的应用程序也有相应的子系统提供支持。各个子系统转入ntdll或者直接转入内核中的ntoskrnl，最终到硬件抽象层hal来实现。这样，设计子系统时，只需要实现对ntdll提供的接口的封装即可。各个子系统最终会转入ntdll.dll或直接调用0x2e中断进入ntoskrnl.exe内核态。

作为一个例子，来看一下常见的kernel32!CreateFileA函数最终是如何转入系统层的。做了一些转换工作后，又转向了CreateFileW。

[image]

CreateFileW又转发到ntdll!NtCreateFile。

[image]

ntdll!NtCreateFile如下：

[image]

在Windows 2000以后的系统里，ntdll!NtCreateFile是如下样式：

[image]

在Windows XP之后，系统用Sysenter进入内核，不会主动使用int 2e作为调用内核函数的方法，不过int 2e仍然被保留下来，还可以使用它们调用Native函数而无须通过ntdll。

不只是ZwCreateFile，ntdll!NtXxx大部分函数（除了NtCurrentTeb）都是由这样一个被称为Stub的函数转向系统层中ntoskrnl中真正的函数的。ZwXxx/NtXxx这些函数在ntdll中看起来只有第一行指令“moveax,XXX”的XXX不相同，这里的XXX是一个索引，用来在内核中定位真正的函数。例如：

[image]

稍后研究Stub与索引的问题。继续分析NtCreateFile，它将eax设置为25h后又调用了KiFastSystemCall。代码如下：：

[image]

这条Sysenter指令OllyDbg跟踪不下去了，CPU执行这条指令之后就会进入内核态，而OllyDbg只是一个用户级调试器。这条指令是Windows XP之后的系统才使用的，关于Sysenter和对应的Sysexit以及将要提到的rdmsr和wrmsr指令，请看《P4_IA32 Intel Architecture Software Developer's Manual》以及wowocock写的《SYSENTER简介及相关例子》。简单地说，Sysenter指令会设置一系列环境转入MSR的SYSENTER_EIP_MSR寄存器中的地址执行，这个寄存器号为176，可以用WinDbg输入“rdmsr 176”得到，更详细的资料请参考《XP下sysenter hook-RDMSR-WRMSR》。在WinDbg中输入命令：

[image]

可见转入了地址80541770，用ln命令可以看出这个地址对应什么函数：

[image]

这样那些Stub函数最终进入了KiFastCallEntry。刚才说到Windows XP下仍然可以使用int 2e，现在就来看看为什么。中断2e的处理函数（ISR）是KiSystemService，用lkd命令“!idt - a 2e”可以看到：

[image]

也就是说，int 2e指令会执行KiSystemService，这个函数做了一些准备工作又会转到KiFastCallEntry中，所以Sysenter和int 2e是异曲同工的。

[image]

至于KiFastCallEntry，这个函数比较长，详细一些的分析可参考《Undocumented Windows 2000 Secrets》，ReactOS中也有仿真代

码。这个函数主要做了一些参数检测的工作，最后使用Stub函数中设置的eax作为索引，查找系统中的一个表，表项对应的地址就是内核函数真正的地址，找到之后就可以设置参数调用了。

2．SDT

系统服务表（System Service Table，SST）是被存储在服务描述表（Service Descriptor Table，SDT）中的一个表项，它的结构官方没有公开，不过《Undocument Windows 2000 Secretes》出版了C语言定义：

[image]

一些AntiRootkit软件中也称SDT为系统服务描述表（System Service Descriptor Table，SSDT），以后本书不区分SSDT与SDT。 SDT结构的定义是这样的：

[image]

一个SDT可以包含4个SST，但不是所有的SST都被使用到。ntoskrnl导出了一个名为“KeServiceDescriptorTable”的SDT，它里面只有一个SST对应着ntoskrnl中的函数。用lkd可以看到：

[image]

不过系统还维护着另一个未导出的SDT，叫做“KeServiceDescriptorTableShadow”，这个SDT会指派给GUI线程使用。因为user32.dll和gdi32.dll一些函数也会通过类似于ntdll中的那些Stub函数来调用内核中的代码。

[image]

SST的ServiceTable就是函数的地址表了，与ntoskrnl.exe对应的SST中ServiceTable还可以通过导出的地址KiServiceTable来访问：

[image]

前面跟踪ntdll!ZwCreateFile设置的索引是25，通过这个表，就可以知道它的庐山真面目了。

[image]

正是nt!NtCreateFile，地址是80578ed2h，这个地址大于7fffffffh位于内核，是货真价实的函数，向下就会调用IO函数了。

[image]

在ntoskrnl中，也有ZwXxx和NtXxx两个版本的函数，不过不像ntdll中是同一个函数的不同名字。

3．Zw和Nt

Zw和Nt开头的函数有什么区别？OSR Online上的《Nt vs. Zw - Clearing Confusion On The Native API》将这些问题讲得很清楚，读者可以花点时间去注册一个Member。

ntdll.dll中Zw和Nt两种开头的名字，以ZwWriteFile和NtWriteFile为例，用LordPE查看这两个函数的RVA，会发现这两个名字事实上指向同一段Stub函数，这个Stub函数再通过Sysenter调用ntoskrnl.exe中的函数。得出的结论就是ntdll中ZwXxx与NtXxx是同一个Stub函数的不同名字。

相应的，用WinDbg的lkd观察ntoskrnl中的函数，看看有什么不同：

[image]

可以看到，nt!ZwWriteFile也是一个Stub函数，而nt!NtWriteFile却不像ntdll!NtWriteFile，它是真正的内核函数。内核中的NtXxx是各个Stub函数最终转向的、实现具体功能的函数。如果调用来自用户态，即PreviousMode为UserMode，就会进行一系列参数检测；反之如果调用来自内核态，即PreviousMode为KernelMode，则不会检测参数。

再回顾一下：

（1）ntdll.dll中ZwXxx与NtXxx函数都是导向ntoskrnl.exe的Stub函数。

（2）ntoskrnl.exe中ZwXxx函数也是Stub函数，导向ntoskrnl.exe的NtXxx函数。

（3）ntoskrnl.exe中NtXxx函数是所有Stub的目的地，是真正做事情的函数。

15.2.7　Hook和AntiHook

看似强大的保护通常都有致命的弱点，一些著名的游戏反外挂程序，只要找到一个小小的开关，所有保护特性都会被关闭。

1．Hook

本节所使用的反调试技巧都是使用Native API进行的，如果调用的Native API被人动了手脚，那么一切检测都像是别人手里的提线木偶。例如，OllyDbg调试器插件HideOD可以自动挂钩这些Native API，于是教科书式的反调试都无效了。

下面来看一下HideOD是如何让相应的Native API不能正常工作的。加载目标程序后，在OllyDBG的CPU窗口查看ZwSetInformationThread函数汇编代码。正常的代码如下：

[image]

执行HideOD插件隐藏功能，ZwSetInformationThread将被修改成类似下面这个样子：

[image]

查看插件申请的地址，即[edx]所指向的代码是这样的：

[image]

中间调用了910220，用来过滤掉ThreadHideFromDebugger这个动作：

[image]

通过这种Ring 3级别的简单钩子，调用ThreadHideFromDebugger检测调试器的诡计就无声无息地幻灭了。

2．AntiHook/Splicing

Hook是万能的，不过所有公开的Hook都是无用的。就像前面的例子一样，HideOD插件钩住了ZwSetInformationThread、ZwQueryInformationProcess等。简单的Ring 3钩子Hook了ZwXxx，所有依靠Native API的检测都失效了。

HideOD中的代码是经过改良的，最早期插件直接将ZwXxx函数入口写入一个jmp指令，跳到另一个地方，过滤掉一个危险的操作，或者执行原来的函数，最后返回。Hying的PE-Armor外壳中很早用到了ThreadHideFromDebugger技巧，很多人写了简单的插件甚至手工干脆把ZwSetInformationThread改成了“retn 10”来防止调试器与被调试程序脱钩，于是Hying改进了他的壳，当发现所要调用的函数代码不是以下两种形式时，就引发错误。

[image]

这两种形式正是Stub函数在Windows 2000和Windows XP下的样子，没有哪个导出的Native API代码是其他的形式，除非它被下了断点或者被Hook了。这样一来直接写jmp指令来Hook或者直接“retn 10”的做法行不通了，不过PE-Armor并不对操作系统版本进行严格的检测。如果在Windows 2000下出现Windows XP的Stub也不会有异议，而Windows XP下的Stub函数保持格式的情况下还有修改第2行xxxxxxxx的余地，于是改进型的Hook产生了，就是HideOD中的Hook：

[image]

针对这个改进，自然而然想到的对策就是检测第2行的地址是否属于ntdll，很明显911008不符合要求，不过这样还是不太奏效，因为在ntdll中找一个没有被使用的空间来存放Hook代码很容易，PE Header里、区块间隙中有充足的空间。

这种无休止的小机关战争令人厌倦，俄罗斯程序员PSI_H的《Простой способ противодействия сплайсингу API》（对应的中文翻译《对付API-splicing的一种简单方法》）介绍了对付API Hook更高明一些的技巧。

Splicing这个术语的含义是把一段代码抽走，放到壳动态申请的内存中，然后生成一个跳转指令跳过去执行，这样抓取内存镜像时就会丢掉这一部分，从而使脱壳变得麻烦一些。而现在的Hook程序，比如微软的Detours库，也是将函数的开头部分复制到另一段内存，再把原来的地方写上一个jmp指令跳到一个地方拦截对这个函数的调用，最后再执行复制出来的代码，继而转入原始的函数，所谓的“inline hook”指的也是类似的手段。

这里结合文中的代码介绍一下击败Splicing实现的Hook：

[image]

这段代码恢复了Ring 3级对ntdll.dll中ZwWriteVirtualMemory这个函数的挂钩，因为LoadLibrary去加载一个已经加载过的DLL会直接返回它的HINSTANCE，这里要的是原始的未经修改的函数代码，因此要把DLL复制到另一个地方改头换面进行加载。新加载的DLL没有被修改，因此就可以得到对应函数原始的代码，再把它们写回原来的地址，这样就摘除了Hook。PSI_H的文章还指出：“尽管这里给出的摘除Hook的方法完全奏效，但需要加载新的DLL模块，这可能会引起防火墙的暴怒。所认为的更为优雅的办法就是只需从文件中读取所需要的字节，并且给出了一段例子代码，有兴趣的读者可以参考”。

Themida也是通过读取文件来防止被Hook，不过它做得更“过火”，干脆不恢复钩子直接去调用读出来的代码。而系统DLL，例如user32.dll，总是被加载到同一个地址，因此读出来的代码可以不需要重定位就可以调用，代码所访问的数据地址都是真实DLL中的有效地址，可以跟踪一下softworm《一个小花招》提供的例子程序来体会一下。对付这种完全自己读取DLL代码的防止Hook的防御措施，一般是在程序读取DLL代码用的Buffer中搜索函数的特征码来下断点，可以参考kanxue老师的《如何中断Themida的MessageBox对话框》。

3．小结

像所有的故事一样，事情应该有个结局了。自己读取DLL代码的方法对Hook有一定阻挡作用，可是毕竟LoadLibrary(Ex)甚至Themida用的ReadFile这些函数还是会被Hook（比如deroko写过Themida-Spy）。回忆前面的内容，从Windows 2000到Windows XP都支持使用int 2e来调用Native API，只需要用这样的代码：

[image]

下面这种方式比较隐蔽地调用了ThreadHideFromDebugger：

[image]

这段代码直接在堆栈中设置好参数就通过int 2e进入内核了，所以不管如何修改ntdll.dll中的ZwSetInformationThread都不会对这个反调试代码有任何影响。为什么这段代码调用了4个不同的索引？shoooo解释说，在不同的NT系统上ZwSetInformationThread的SDT索引不一样，

调用4个让这个Anti技巧在不同的操作系统上都有效，可能有人认为用GetProcAddress得到ZwSetInformationThread再读取“mov eax, xxx”的立即数会得到准确的索引。而事实上，从外界获取信息越少越安全，因为如果有人把索引改成另一个函数的索引，会因为参数不合法而不产生任何操作。因此这里我们得到一点不同以往的认识，那就是硬编码并不总是坏的。反过来说，因为4个索引只有一个会“中标”，其他三次调用也会因为参数不合法而不产生操作，因此也不需要太担心这样的蛮力调用会有什么副作用。

这个Anti可以无视所有Ring 3的Hook，将它用虚拟机保护起来，会让许多调试者挠头，却也不是完美的，因为int 2e还是会调用KiSystemService，查找SDT中ntoskrnl中的函数。因此只要写一个驱动，通过修改SDT，或者inline hook ZwSetInformationThread，把过滤器放在内核中，这个努力了一整节的Anti还是被无情地淘汰掉。更简单的隐藏调试器的方案是运行HideToolz，或者为OllyDbg安装一个Phantom插件，它们已经写好了这种驱动程序。

15.3　真正的奥秘：小技巧一览

把任何一个反调试手段单独摆出来，要化解它就如捏死一只小蚂蚁一样，不过请回忆一下童年的画册，成千上万的军团蚁聚在一起，黑压压的一片，所到之处遍地白骨，是多么可怕的景象。事实上，就像hying在《软件加密内幕中》说的，真正令破解者头疼的，并不是设计得多么精妙的一个新奇装置，往往就是那些司空见惯、单调乏味的反调试手段堆砌起来的东西，让天生不喜欢重复劳动的Cracker不得不陷入无趣的体力劳动中，拖垮他们的耐心，才是真正成功的加密。

在本节里，走马观花似地浏览一些五花八门的反调试小技巧，笔者还会介绍一些曾经流行一时却已失去生命力的东西，献给奋斗在那个年代的人。Ap0x用MASM写了很多这方面的例子，笔者会借用。

15.3.1　SoftICE检测方法

SoftICE作为最著名的内核级调试器，早在Windows 9x时代就已成为最流行的调试工具。所以对抗SoftICE的办法早就被大家研究透彻。

1．句柄检测

句柄方法检测原理是用CreateFileA或_lopen函数试图获得SoftICE的驱动程序“\\.\SICE”（Windows 9x版本）、“\\.\NTICE”（Windows NT版本）等的句柄，如果成功，则说明SoftICE驻留在内存中。这种方法也称为MeltICE子类型。

但DriverStudio 2.x以后的版本用这种方法不能检测到。该系列的Symbol Loader检测SoftICE是否激活方法是先将DriverStudio安装序列号取出，经过简单的运算，得到4个字符“xxxx”，然后将“\\NTICE”与“xxxx”连接成“\\NTICExxxx”，最后用CreateFile("\\NTICExxxx ",...)来检测到SoftICE是否激活。详见nmtrans.dll中NmSymIsSoftICELoaded函数。

这种方法还可以用来检测Filemon、Regmon等工具。

2．BoundsChecker后门

Numega公司除了出品了SoftICE外，还给开发者提供了一个内存检测工具，那就是BoundsChecker，利用它可以检测内存泄露、资源泄露等程序开发中常见的错误，SoftICE为BoundsChecker留了一个后门接口。

[image]

3．SoftICE后门指令

后门指令（Back Door Commands）通过中断INT 03来进行。DOS时代，用后门指令可以获得SoftICE版本信息、设置断点和执行命令等。当然这些技术都已过时了，现在这些后门中可能只有一条RET指令了，什么都不做。

执行SoftICE命令的INT 03子功能描述如下：

[image]

每个INT 03子功能的入口参数都有相同的部分，即SI = 4647h('FG')；DI = 4A4Dh('JM')。这两个入口值在SoftICE中叫做“魔法值（Magic values）”，凡是SoftICE的后门指令都必须以这两个数为标志。

后门指令现在主要是被用做检测SoftICE，这种方法要结合SEH来实现，否则当SoftICE不存在时就会引起一个断点异常。

4．判断NTICE服务是否运行

在Windows NT/2000/XP系统中，SoftICE是一个内核设备驱动类型的服务，服务名称是NTICE，因此可通过判断NTICE服务是否运行来检测SoftICE。

5．利用UnhandledExceptionFilter检测

SoftICE作为系统级调试器，会把自己置为系统默认调试器并捕获系统异常，其做法是在载入时，会把kernel32!UnhandledExceptionFilter的第一字节“55”用“CC”来代替。因此就可以根据这个“CC”机器码，判断SoftICE是否加载。

6．int 2d

int 2d本来是被内核ntroskrnl.exe运行DebugServices用的，但是也可以在Ring 3模式下使用它。如果在一个正常应用程序中使用int 2d，将会发生异常，然而如果这个程序被附加了调试器，就不会产生异常。

[image]

int 2d不仅能够用来检测Ring 3下的调试器，同时也能检测DbgMsg驱动，这意味着，它可以用来检测Ring 0下的SoftICE。

除此以外，int 2d还有个妙用，因为附加调试器的程序在运行完int

2d后，会跳过此指令后的一个字节。

[image]

如果在调试器中步进或步过int 2d，不同的调试器会有不同的执行方式，所以可以用int 2d来完成一些代码混淆工作。这部分不是本章内容，有兴趣的朋友可以研究一下。

15.3.2 OllyDbg检测方法

如果让所有的解密者都只能选择一个工具，大部分的人都会选择OllyDbg。OllyDbg非常的强大，对于它的检测变得很重要。

1．查找特征码

想想在生活中，怎样识别两张近似的脸孔？是的，根据特征！同样的，特征检测在计算机领域也被广泛应用。比如，杀毒软件就是根据特征码来辨识病毒的，庞大的病毒库里，包含的最主要内容就是形形色色的病毒特征码，这些特征码就是从病毒体内不同位置提取出来的一系列字节。对于OllyDbg的检测，也可以采用类似方法。

例如，对OllyDbg 1.1版本提取特征码：

（1）地址：401126h　特征码：83 3D 1B 01

（2）地址：43AA7Ch　特征码：8D 8E 83 21

当程序在运行时，对当前运行的所有进程做一次枚举，如果发现某进程的目标地址有这个特征码，就可以认定是监测到OllyDbg了。

[image]

为了降低误报率，还可以多检测几个特征码。值得注意的是，这种方法只能检测某一个或某几个版本的OllyDbg。由于不知道对手使用的是什么版本的OllyDbg，所以这种方法并不能确保检测结果。

2．检测**DBGHELP**模块

调试器一般用Microsoft提供的DBGHELP库来装载调试符号，如果一个进程加载了DBGHELP.DLL，那么它很可能是一个调试器。以用CreateToolhelp32Snapshot创建进程的模块快照，通过Module32First和Module32Next来枚举模块，看看是否有不良之辈。

但是这种检测是脆弱的，只需要把DBGHELP改名，再修改OllyDbg中对应的名字字符串，它就失效了。

3．查找窗口

不要忘记，Ring 3调试器，也是一个普通的Windows程序而已。所以，可以用两种常见的方式去检测OllyDbg：查找窗口和查找进程。

查找窗口，可以用三种方法：

（1）FindWindow

像所有的对话框一样，OllyDbg的主窗口，也有它的标题和类名。使用这个API函数可以判断OllyDbg的主窗口是否打开。既可以通过类名来查找窗口，也可以通过标题来查找，如果要搜索子窗口，需要使用

FindWindowEx。

（2）EnumWindow

这个函数枚举所有顶级窗口，并调用指定的回调函数，可以在回调函数中用GetWindowText得到窗口的标题，比较是否包含“OllyDbg”字样。

（3）GetForeGroundWindow

这个函数与前两种方法略有不同，它并不枚举窗口，而是返回前台窗口（用户当前工作的窗口）。系统分配给产生前台窗口的线程一个稍高一点的优先级。

当程序正在被调试时，调用这个函数将获得前台窗口，也就是OllyDbg的窗口句柄。

4．查找进程

枚举进程检测是否有OllyDbg.exe进程存在。这个方法和查找窗口方法一样，都很好被跳过，调试者只要把OllyDbg稍作修改，改进程名字和标题名字就可以了。

5．SeDebugPrivilege方法

在默认情况下，进程是没有SeDebugPrivilege权限的。然而进程通过OllyDbg和WinDbg之类的调试器载入的时候，SeDebugPrivilege权限被启用了。这种情况是由于调试器本身会调整并启用SeDebugPrivilege权限，当被调试进程加载时SeDebugPrivilege权限也被继承了。

可以通过打开CSRSS.EXE进程间接地使用SeDebugPrivilege确定进程是否被调试。普通程序的默认权限，是无法对CSRSS.exe执行OpenProcess的，如果能够打开CSRSS.EXE，则意味着进程启用了SeDebugPrivilege权限，由此可以推断进程正在被调试。这个检查能起作用是因为CSRSS.EXE进程安全描述符只允许SYSTEM访问，但是一旦进程拥有了SeDebugPrivilege权限，就可以忽视安全描述符9而访问其他进程。注意：在默认情况下，这一权限仅仅授予了Administrators组的成员，也就是说，如果用户以非管理员身份登录，该检测方法将失效。

下面是SeDebugPrivilege检查的例子：

[image]

如果OpenProcess()成功，则意味着SeDebugPrivilege权限被启用，这也意味着进程很可能被调试。

6．SetUnhandledExceptionFilter方法

这个方法和异常处理有关，当一个异常未被任何SEH处理而到达Unhandled Exception Filter（kernel32!UnhandledExceptionFilter）并且程序没有被调试时，Unhandled Exception Filter将会调用kerel32!SetUnhandledExceptionFilter作为参数指定的高层exception Filter。如果被调试，则把异常发往调试器。可以利用这一点，通过设置exception Filter然后抛出异常，如果程序被调试，那么这个异常将会被调试器接收；否则，控制被移交到exception Filter运行得以继续。利用

SetUnhandledExceptionFilter检测调试器原理和ProcessDebugPor类似，看雪论坛simonzh2000写过一篇很详细、很完整的文章。

下面的示例中通过SetUnhandledExceptionFilter设置了一个高层的exception Filter，然后抛出一个违规访问异常。如果进程被调试，调试器将收到两次异常通知；否则，exception Filter将修改CONTEXT.EIP并继续执行。

[image]

有些程序是直接通过kernel32!_BasepCurrentTopLevelFilter手工设置exception Filter的，这样做更加隐蔽，以防破解者在那个API上下断。

7．EnableWindow方法

这是一个小小的伎俩，却往往可以让Ring 3调试器中招。调用这个API可以暂时锁定前台的窗口，让用户休息一下，顺便也让调试器无法工作。

[image]

处理完之后，当然要恢复用户的窗口了，要做得更“狠毒”一些，可以创建线程不断锁住所有的窗口。

8．BlockInput方法

跟EnableWindow大同小异的小伎俩，调用BlockInput(TRUE)可以锁住键盘，完成工作后用BlockInput(FALSE)恢复。这种锁定可以按“Ctrl + Alt + Del”键强制解除。

15.3.3　调试器漏洞

软件在与破解者的对抗中，并不是总处于被动防守的姿态，有的时候，攻击才是最好的防守方式。攻击调试器成为现在Anti-Debug技术中的重要一环。

只要是软件，就存在漏洞，调试器是软件中的一种，自然也不能例外。发现和利用它们自身的漏洞来攻击往往是非常有效的。下面以OllyDbg为例，列举一些已知的漏洞。

1．OutputDebugStringA

OutputDebugStringA函数用于向调试器发送一个格式化的串，OllyDbg会在底端显示相应的信息。OutputDebugString漏洞本质上是一个格式化串的溢出漏洞。格式化串其实也是很严重的漏洞，轻则崩溃，重则可以导致执行任意代码。OllyDbg的问题就是对格式化串过滤不严间接导致了缓冲区溢出的发生，保存在栈中的返回地址被覆盖。其实，以下库函数都存在格式化串溢出漏洞printf，fprintf，sprintf，snprintf，vfprintf，vprintf，vsprintf，vsnprintf。

先来看一个简单的例子：

[image]

程序打印出自己的参数。如果输入“hello world”，则输出“hello world”。但是如果输入“%d”，会发现程序输出“4198693 ”，十六进制就是401125。正常地打印一个十进制数值应该是带参数的，比如printf("%d",i)。i就是一个整型变量。这里略了后者，当所有参数压栈完毕调用printf函数的时候，printf并不能检查参数的正确性，只是机械式地从栈中取值作为参数，也就是看到的4198693，这个时候堆栈就被破坏了，栈中的信息就泄露了。

尽管OllyDbg已经对OutputDebugString输出的字符串进行了长度检查，只接受255个字节，但是没有对提供的参数进行检查，所以间接导致了缓冲区的溢出。可以将参数设置为"%s%s%s "，调用OutputDebugStringA，会让OllyDbg崩溃。

当然，经过精心构造的输出串，使OllyDbg溢出后执行任意代码也是完全可以做到的，这里不再详述。一些修改版本的OllyDbg已对这个漏洞进行了修补。

2．DRx清理BUG

OllyDbg只要捕获到被调试程序的异常，就毫不留情地把所有DRx(DR0-DR7)清零。这是一个不能称之为bug的bug，但是却可以被利用。在异常中设置DRx的值，再利用这些值进行解码，已经成为了很多壳的手段。运用最成熟的，当属Hying's PEArmor壳。

PE-Armor壳设置了很多SEH，并在SEH中改写DRx的值，利用这些值进行解码，才能使程序正确地运行下去。如果使用OllyDbg遇到异常会清掉DRx，用DRx值来解码就会出错。

15.3.4　防止调试器附加

防止调试器附加（Anti-Attach）是非常重要的，因为在不方便使用调试器启动程序的时候，调试者往往先运行目标程序，再使用调试器附加到目标进程（例如调试游戏时）。Ring 3调试器的附加使用的是DebugActiveProcess函数，在附加相关进程时，会首先执行到ntdll.dll下的ZwContinue函数，最后停留在ntdll.dll的DbgBreakPoint处。熟悉OllyDbg的朋友一定对这个函数不陌生。事实上是调试器在这里设了一个int 3断点，由调试器自己捕捉。当调试者按F9键继续后，调试器再恢复这里的代码继续运行。

只要在这两处调试器Attach过程中设置一点点障碍，就能有效地阻止程序被附加调试。先看看下面这个例子：

[image]

[image]

[image]

[image]

这段代码挂接了Ntdll.ZwContinue，如果经过这里，则报告发现调试器；如果未经过这里，则说明程序没被附加调试。而对DbgBreakPoint函数，也可以采用同样的方式检测，或者可以直接在程序中检测这里是否有int 3，也可以达到同样的目的。

15.3.5 父进程检测

理论上来说，当一个程序被正常启动时，它的父进程应该是Exploer.exe（资源管理器启动）、cmd.exe（命令行启动）或者Services.exe（系统服务）中的一种，当某进程的父进程并非上述三个进程之一时，一般可以认为它被调试了（或者被内存补丁之类的loader加载了）。

下面是实现这种检查的一种方法：

（1）通过TEB（TEB.ClientId）或者使用GetCurrentProcessId来检索当前进程的PID；

（2）用Process32First/Process32Next得到所有进程的列表，判断explorer.exe的PID（通过PROCESSENTRY32.szExeFile）和通过PROCESSENTRY32.th32ParentProcessID获得的当前进程的父进程PID；

（3）如果父进程的PID不是explorer.exe、cmd.exe或Services.exe的PID，则目标进程很可能被调试。

需要注意的是，在一些非正常启动进程情况下，例如某进程启动另一个进程时，这个调试器检查会引起误报。

15.3.6 时间差

一个程序被断点打断，CPU捕获异常发给调试器，调试器处理后继续运行，这个时间显然比程序直接执行要慢很多，计算一个操作开始和结束运行的时间，如果慢得不合理，那么可以判断被跟踪了。

RDTSC指令（Read Time-Stamp Counter）用来获得CPU自开机运行的时钟周期数。它的结果是64位的，保存到eax和edx两个寄存器中，本来设计为用来精确测量算法开销，不过现在利用两次使用该指令来计算时间差。

[image]

当然，也可以采用kernel32!GetTickCount()实现类似功能的检测。

15.3.7 通过Trap Flag检测

CPU符号位EFLAGS中一位叫做TF，即TRAP FLAG。当这个TF = 1的时候CPU执行完EIP的指令就会触发一个单步异常。例如这段代码：

[image]

15.3.8 双进程保护

Windows下Ring 3调试器对被调试程序的关系是“一个萝卜一个

坑”，一个进程只能有一个调试器。熟悉脱壳的朋友一定不会对Armadillo双进程保护功能陌生，它所使用的就是这种技术。

只需要对Windows提供的调试API稍加了解，就能够对自己的软件产品实行双进程保护了。一个简单的调试器应该实现以下一些功能：

（1）加载一个进程或附加到一个正在运行的进程上

使用CreateProcess创建进程时，指定DEBUG_PROCESS标志，来启动被调试进程，或者使用DebugActiveProcess函数绑定到某个正在运行的进程上。

（2）获得被调试程序的底层信息，包括进程ID、映像基址等

使用WaitForDebugEvent等待调试事件发生，该函数阻塞调用线程直到等待到调试信息。

（3）接收被调试进程发来的调试事件并进行处理

当WaitForDebugEvent返回时，意味着在被调试进程中发生了调试事件，响应并处理该调试事件，继续执行被调试程序。

关于Windows调试API并非本章重点，这里只是略加阐述以便于理解双进程保护的机制，有兴趣的朋友可以自己实现一套自己的双进程保护系统。

第16章　外壳编写基础②

对一个程序文件加密有一个方面是必须要考虑的，这就是防止解密者对程序文件的非法修改和反编译。要实现这种保护最常用的方法之一就是给编译好的程序文件加上一个外壳。

对程序的加壳一般是选用一款现成的加壳软件。使用现成的加壳软件虽然很方便，但却存在着一些不可避免的缺点：越是先进、优秀的加壳软件有时反而会越不安全。为什么呢？因为加壳软件越优秀，用它加密的软件越多，研究它的人也会越多，其中必定有一些高手会分析出这些外壳所用的关键技术，并将其公开，有时甚至会针对这些加壳软件而写出专门的脱壳机。一旦一个加壳软件被写出脱壳机，那用它加密的软件的保密性就可想而知了。所以写一个自己专用的加壳软件还是有一定意义的。

16.1　外壳的结构

本章所讲述的加壳工具由两部分组成，第一部分是主体程序，主要是将原PE文件读入到内存，然后对PE文件各个部分加工，主要是各区块数据压缩，将输入表、重定位变形，最后将外壳部分与处理好的主体文件拼合。第二部分就是外壳部分，这部分主要是加壳后程序执行时候的引导段，它模拟PE加载器处理输入表、重定位表，最后跳到原程序执行。

最终装配成一个完整壳的程序结构如图16.1所示。目标程序新增了一个区块“.pediy”，这部分就是“壳”。区块.text、.data等是原始程序的代码数据，不过其现在是以压缩的形式保存的。另外，为了简化，本例没处理输出表，其所在区块.edata仍以明文形式存在。在“.pediy”区块里，以ShellStart为分界，之前的部分以非压缩的方式存在，之后的部分是以压缩的方式存在的。新程序的入口点指向外壳ShellStart0开始的部分，外壳执行时先执行这部分，这部分的主要功能是将ShellStart开始的真正的外壳代码在内存中解压缩，并初始化一些数据。初始化完成后转移到ShellStart继续执行。ShellStart开始的代码是外壳的真正部分，它的主要功能是还原原始程序（.text、.data等区块数据），另一个重要功能则是阻止破解者的跟踪、脱壳。所以一般来说这段代码会比较长，里面还有各种反调试器、反Dump的代码。将它以压缩的方式存储一方面可以减小文件尺寸，另一方面也有利于程序的安全性。

[image]

图16.1　加壳后程序的结构图

16.2　加壳主程序

本章将通过一个实例，说明如何编写一个简单的加壳程序。在这个加壳程序中要实现的主要功能有：对程序的压缩、对资源的处理、对输

入表的处理、对重定位表的处理、区块的融合和额外数据的保留等。为了突出重点，在此实例中基本不涉及对各种常用调试工具的防范，需要此类功能的读者，可以参考本书中的其他相关章节，添加各种相应的代码。

为了使源程序更易于讲解方便，主程序采用Microsoft Visual C++ 6.0编写，外壳部分采用汇编，因此需要读者有一定的汇编编程基础。完整的源码在配套光盘映像文件中。

16.2.1 判断文件是否为PE格式

本节程序处理的加密对象是EXE和DLL文件，所以在对文件进行处理前必须先判断目标文件是否为正确的PE格式。在本例中，文件格式的判断是通过使用一个自定义名为IsPEFile()的函数来进行的。如果格式符合，函数返回1，同时在消息框中输出格式正确的消息；否则返回0，并输出相应的错误消息。

校验的方法是先检验文件头部第一个字的值是否等于IMAGE_DOS_SIGNATURE，也就是字符串“MZ”，如果是则表示DOS MZ header有效。其次根据e_lfanew字段找到PE header，校验比较PE header的第一个字的值是否等于IMAGE_NT_SIGNATURE，也就是字符串“PE”。如果前后两个值都匹配，那就认为该文件是一个有效的PE文件。

通过校验FileHeader结构中Characteristics字段的值，判断是EXE文件还是DLL文件。特征值IMAGE_FILE_DLL表示该文件是DLL。在WINNT.H中已经被定义：

[image]

实现检测的流程如下：

（1）判断文件开始的第一个字段是否为IMAGE_DOS_SIGNATURE，即5A4Dh。

（2）通过e_lfanew找到IMAGE_NT_HEADERS，判断Signature字段的值是否为00004550h，即IMAGE_NT_SIGNATURE，如果是IMAGE_NT_SIGNATURE，就可以认为该文件是PE格式。

（3）判断该PE文件是否可加壳。如果该文件只有一个区块就认为是被加壳了，或其入口点的值大于第二个区块虚拟地址值也认为其被加壳了。

（4）最后，校验一下FileHeader结构中Characteristics字段的值，判断是EXE文件还是DLL文件。

16.2.2 文件基本数据读入

文件格式判断正确后就可以将文件读入内存等待处理。文件的读入方式一般有两种：一种是直接按文件偏移的方式读入；另一种是依据PE文件的结构，仿照Windows装载器载入PE的方式，根据各个区块的RVA分别读入。

第一种读入方式编程时非常简单，只需利用GetFileSize函数取得欲处理文件的大小，然后申请一块同样大小的内存，再一次性读入整个文

件即可，或者利用CreateFileMapping和MapViewOfFile将文件映射到内存也可以。读入后要使用文件中的某个数据时，只需要根据它的文件偏移（Offset）加上读入基址就可以找到这个数据，上一节的IsPEFile()函数就是采用这种方式。这种方式虽然读入操作简单，但是在以后的处理中却有很大的弊端。因为在PE文件中，所有的数据都是根据RVA或VA来定位的，如果要使用一个根据RVA来定位的数据，那就先得把它的RVA转换为Offset，然后才能找到并使用它。如果要处理的数据比较多且分布在不同区块中的话，这种转换就会显得非常麻烦，而且稍有疏忽就容易引起错误。所以笔者建议使用第二种读入方式。

第二种读入方式是先取得PE文件头的SizeOfImage字段的值，然后申请相应大小的内存，再根据文件的Section Table中的VirtualAddress和VirtualSize的值逐个区块分别读入。如此操作，要使用数据时只需根据数据的RVA加上读入基址就可找到并操作它了。

先定义几个全局变量，将读入内存的映像相关参数放入，方便以后随时读取。

[image]

下面就是文件读取所使用的代码：

[image]

[image]

[image]

16.2.3 附加数据读取

某些特殊的PE文件在各个区块的正式数据之后还有一些额外数据。这些额外数据不属于任何区段，所以当程序被Windows装载器载入时它们不会被直接读入内存，而是事后由程序在需要使用时自行读取。这些额外数据对于程序的运行一般是至关重要的，但是按照上一小节的方法将文件读入内存时，这些数据不会被读入。所以当加密完成重写文件时它们可能会丢失，造成程序无法运行。对于这类程序，必须在读入文件时将这些额外数据单独读取、保留，等待加密完成后再追加到文件的最后。

额外数据的起点可以认为是最后一个区块的末尾，终点是文件末尾，所以额外数据的大小就是文件大小减去文件头到最后一个区块的末尾的大小。

读取额外数据所使用的代码如下：

[image]

16.2.4 输入表处理

在如今常见的那些带加密功能的外壳中，破坏原程序的输入表几乎是必有的功能，而且是各出奇招，尽可能让脱壳者无法修复。

要破坏一个程序的输入表一般要有两步。第一步是在加密时做的，它破坏原程序中的输入表，将其换一个形式存储。但是仅此还不够，如果外壳初始化原程序时仍将各个函数的正确地址写回输入表，那么借助某些工具就可以轻易地重建一个可用的输入表，所以破坏输入表还需要第二步。

破坏输入表的第二步就是外壳对原程序进行初始化时不把真正的函数入口地址写回输入表，写回的是外壳中一段程序的入口，通过那段程序的变换后，再转到正确的函数入口去执行。这样通过中间的一些变换，就可以欺骗一些重建输入表的软件。如果不能重建输入表的话，脱壳也就失败了。在本例中只研究破坏的第一步，至于第二步读者可以参考其他一些源码。在看程序之前，先来分析一下程序正常载入时输入表的初始化过程。

首先系统根据输入表项中的Name字段找到DLL名，根据DLL名获取DLL在内存中的句柄，然后再根据OriginalFirstThunk字段找到IMAGE_THUNK_DATA结构，它一般是指向IMAGE_IMPORT_BY_NAME的指针数组，或者也可能是函数在DLL中的序列。根据函数序列或IMAGE_IMPORT_BY_NAME，就可以得到函数的入口地址，再将获取的这些入口地址写回到FirstThunk指向的IMAGE_THUNK_DATA结构数组就可以了。如果OriginalFirstThunk为零，则用FirstThunk代替。

由此可知，在转储后的输入表结构中只要包含了FirstThunk也就知道了要初始化的数据的地址，知道了DLL名和函数名或函数序号也就知道了要填这些地址的函数入口，知道这些就可以完成对原程序输入函数的初始化了。据此，笔者设计了如图16.2所示的新输入表结构。

[image]

图16.2　新输入表结构

在此结构中每个字符串前的一个字节中保存了该字符串的长度，其后为“00”的字节表示了字符串的结束，在函数名字符串前的那个字节中如果存储的是“00”，则字符串中不是函数名而是函数序号。当然，读者也可以设计自己的新输入表结构，甚至可以比笔者的更简单，只要能正确完成原输入表的初始化就可以了。

转储输入表所用的代码如下：

[image]

[image]

[image]

程序的输入表转储完成后，应该将原来的输入表清除，以防止被脱壳者利用。使用的代码如下：

[image]

[image]

16.2.5　重定位表处理

在本例中重定位数据的去除是利用一个自定义函数ClsRelocData()来完成的。它的功能是找到重定位数据所对应的区块，将区块的文件大小改为0，同时将该区块所有数据清零。

下面是程序代码：

[image]

对于一般的EXE文件，它的实际载入基址与优先载入基址一般是相同的，所以它的重定位数据一般是无用的，可以把它去除，这可以使文件更小。但是对于DLL的动态链接库文件来说，Windows系统没有办法保证每一次DLL运行时提供相同的基地址，这样“重定位”就很重要了，重定位数据一般是必需的。为了保证程序能运行起来，外壳必须模拟PE加载器的重定位功能，对相关代码重定位，否则原程序中的代码是无法正常运行起来的。

为了提高壳的强度，将原始的重定位表换个形式存储，外壳程序运行时会根据这个结构重定位相关代码。转储的新重定位结构如下：

[image]

• type：重定位表的类型，由于是讨论i386架构情况，本例仅考虑了TypeOffset数组的类型为IMAGE_REL_BASED_HIGHLOW的情况。

• FirstTypeRVA：这一组重定位数据的开始RVA加上TypeOffset数组第1项的低12位（ItemOffset值）。

• nNewItemOffset：是一个数组。数组大小每项为1个字节。每一项的值是当前的ItemOffset值与上一项的ItemOffset差值。

在转储后的重定位表结构中，FirstTypeRVA指出了第一项重定位的地址，以后每项在这个基础上加上差值nNewItemOffset，依次定位到所有的重定位地址。如图16.3所示是处理前后的重定位表结构的一个样例，这样的结构不光提高了壳的强度，还减少了重定位表数据，缩小了文件体积。

[image]

图16.3　处理前后的重定位表结构

当然，读者也可以设计自己的重定位表结构，只要能正确重定位指定代码就可以了。转储重定位表所用的代码如下：

[image]

[image]

[image]

这段代码将目标文件的重定位处理好后，再放回重定位表所在的地

方。一些加壳软件没有重定位表转储这一步，脱壳后，完整的重定位表就在文件里，如tElock、ASProtect等。

16.2.6 文件的压缩

如果加壳时用到了压缩技术，那么在解密之前还有一道工序，当然是解压缩。这也是一些壳的特色之一，比如说原来的程序文件未加壳时为1～2MB大小，加壳后反而只有几百KB。

一般现在的加壳软件都具有压缩功能，压缩后的程序比较小，便于交流，而且保密性也比较好。

压缩算法是采用现有的压缩引擎，压缩引擎在选择上除了考虑压缩率外，更关键的是其解压速度要快。因为，解压速度快，加壳程序加载速度才不至于受太大的影响。加壳软件一般采用较多的压缩引擎有aPlib、JCALG1、LZMA等。aPlib引擎对于小文件压缩效果较好；JCALG1具有更为强劲的压缩效果，对大文件的压缩效果好；LZMA是7-Zip程序中7z格式的默认压缩算法，具有很高的压缩比。

在本例中压缩引擎使用的是公开的aPlib压缩函数库（www.ibsensoftware.com），读者可以到此下载到包含有多种编程语言使用资料的完整压缩包。VC中调用aPlib很简单，只需包含aplib.h库即可调用aPlib相关函数了。压缩前调用函数m_nSpaceSize估算出需要的内存空间大小，然后调用用aP_pack()函数压缩数据，该函数最后两个参数是回调函数，主要用于配合显示压缩进度条。压缩代码如下：

[image]

PE文件的压缩一般是按区块进行的，这样可以较好地保持原始程序文件的结构，方便程序的载入与还原。但是有些区块由于其功能的特殊性，当程序被系统载入时里面的数据会被系统所使用，所以是不能压缩的。有些则要做一些特殊处理后才能压缩。常见的此类区块有资源块（.rsrc）、输出块（.edata）等。其中由于资源区块处理方法的特殊性，因此将它作为一小节，在下面单独讲解。

在本例中程序的压缩是通过一个自定义函数来做的。具体函数如下：

[image]

[image]

[image]

[image]

16.2.7 资源数据处理

一般程序的资源都集中在资源区块（.rsrc）中，对于那些资源格式比较特殊的程序文件，由于它们不具有普遍性，在此就不去讨论了。

资源区块的大致结构先是资源目录，紧接在后面的才是资源数据。

系统必须根据资源目录才能找到正确的资源数据。一般的资源只有在程序真正运行时才会被用到，但是有些特殊类型的资源却不是这样，即使程序没有运行，它们也可能会被系统读取、使用。例如，当使用“我的电脑”查看某个文件夹的内容时，该目录下的所有应用程序的图标都会被显示出来。这些图标就是程序资源的一部分，它们在程序没有被执行时仍然会被系统读取。这种特殊的资源类型通常包括：Icon（图标）、Group icon（组图标）、Version information（版本信息）等，它们一般是不能压缩的，因为它们一旦被错误处理，轻则产生程序异常如丢失图标，重则可能根本无法运行。另外，由于系统必须根据资源目录才能找到它们，所以资源目录也不能压缩。因此，对于资源区块的压缩一般需要分成三步来做：

（1）找到资源目录和资源数据的分割点，也就是资源数据的起点，在这之前的资源目录部分不能被压缩。

（2）把那些不能压缩的资源类型如图标等从资源数据中提取出来，转移到一个不被压缩的部分，一般是在外壳数据中找一段空间存放它们。同时还要修改资源目录项中相应的指针，以保证当它们被移动后系统仍然可以找到它们。

（3）压缩资源区块中的资源数据。

下面就这三步分别来分析。

第一步使用一个自定义函数来完成，通过这个函数将找到的资源数据起点RVA地址作为返回值放入EAX。函数所依据的原理是遍历所有的资源数据项，比较它们的起始RVA，找出最小的那个作为整个资源数据的起点。函数中涉及资源目录结构的部分，读者可以参考前面章节或MSDN里的资料。函数代码如下：

[image]

[image]

[image]

第二步也使用一个自定义函数来完成。通过这个函数将特定类型的资源移动到指定的位置。资源类型的指定是根据资源ID号来确定的，Icon的ID号为03h，Group Icon为0Eh，Version Information为10h。只要在入口参数中分别指定不同的ID号，就可移动不同类型的资源。为移动三种不同的资源，此函数将被执行三次。函数代码中涉及外壳结构的部分请参考图16.1。函数代码如下：

[image]

[image]

[image]

[image]

在第三步中资源区块的压缩还需要分两步来做。先是将区块中不能压缩的资源目录部分写入文件，然后从资源数据的起点开始压缩资源区块的剩余部分。压缩完后将压缩得到的数据也写入文件。最后再将两部分写入的数据作为一个完整的区块，根据它的大小、偏移等情况修正文件头区块表中的相应信息。完整的代码如下，使用时只需将它们插入上一小节中省略的部分处即可。

[image]

[image]

16.2.8　区块的融合

在PE文件中的各个区块的长度都必须是FileAlignment值的倍数，不足部分则以“00”填充，所以各个区块之间必定都有一定长度的间隙。所谓区块融合就是将一些可以压缩的区块合并为一个区块，然后进行统一处理。这样可以减少压缩后区块的个数，也就减少了区块间隙的个数，这可以减小加壳后文件的体积。要注意的是此处的融合并非是仅仅将各个区块间的空隙挤掉，将数据连接起来，那样做会导致程序中用来定位数据的地址信息变得无效，最终导致程序被破坏。要做的仅仅是将多个区块在逻辑上作为一个区块来考虑，而它们在内存中未压缩时的存储方式（各数据的RVA）并没有丝毫的改变。此处的融合和编译、连接程序时用“/MERGE”参数进行的区块合并是完全不同的概念。

由于是按文件区块的RVA读入文件的，与程序运行时由系统装载器载入的情况基本相同，所以只需要修改文件头中区块表的数据就可以了。为了简化，在本例中仅仅融合最初的几个区块，具体做法是根据区块名从第一个区块（一般是代码块）开始，找到第一个不能压缩的区块，将它们之间的区块合并（包含第一个区块，但不包含最后那个不能压缩的区块），作为一个区块。合并后的区块的RVA为参与融合的第一个区块的RVA，VirtualSize为参与融合的各个区块VirtualSize之和。融合完成后还需将后面没有融合的区块的区块表向前移动，紧接第一个区块的区块表排列。具体代码如下：

[image]

16.3　外壳部分编写

壳和病毒在某些方面比较类似，都需要比原程序代码更早地获得控制权。壳修改了原程序的执行文件的组织结构，从而能够比原程序的代码提前获得控制权，并且不会影响原程序的正常运行。外壳除了还原原始程序外，另一个重要功能则是阻止破解者的跟踪、脱壳。所以一般来说这段代码会比较长，里面还有各种花指令、反调试器、反Dump的代码。因此外壳部分是壳开发的重中之重。

16.3.1　外壳的加载过程

Windows的PE加载器加载可执行程序时，首先根据输入表获取所有

API调用的地址，并填写到IAT中，再重定位所有的重定位项，最后调用WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,PSTR szCmdLine, int iCmdShow)执行；如果是DLL，则调用DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)。加壳后，这个加载过程就由外壳来模拟了。

这里简单说说一般壳的装载过程。EXE和DLL两种方式加壳处理的基本原理是一样的，但也有区别。DLL涉及多次进入、入口参数和返回地址的保存、重定位项处理。

1．保存入口参数

加壳程序初始化时保存各寄存器的值，外壳执行完毕，再恢复各寄存器内容，最后再跳到原程序执行。通常用pushad/popad、pushfd/popfd指令对来保存与恢复现场环境。

2．处理多次进入

由于DLL加载过程中，会多次进入DllMain()，在外壳程序中的一些变量在第一次进入时，就已经初始化。所以，还要有一个记录这些变量是否已经被初始化的标志，以防止重复初始化。举个例子说明如下：假设外壳程序中有一个内存变量“test dd SomeAddr”，而SomeAddr是一个需要外壳重定位的地址，那么在第一次进入时，外壳将对test进行重定位“add test,ebp”，在该DLL并未卸载的情况下二次进入时，又要执行一遍“add test,ebp”，很显然，这时候的test的内容将是错误的。例程中使用“is_data_initialized”这个变量作为标志。

另外，再定义一个变量“S_FileIsDll”来记录当前加载的PE是否为一个DLL，只有是DLL文件时，才进行一些特殊处理。

3．模拟**PE**加载器完成相应的功能

外壳首先将原始数据解压出来，然后模拟Windows系统的PE加载器，恢复输入表。如果有重定位表，则对代码进行“重定位”操作。处理完跳到原始入口点，从这个时候起壳就把控制权交还给原程序了。

16.3.2　自建输入表

在正常情况下，程序通过PE输入表可以得到API函数的地址，从而使程序能够正常运行。那么当程序加壳后，外壳程序中所调用的API函数在执行时又是如何取得其在系统中的地址的呢？

通常加壳程序都是通过自建输入表、壳程序本身再将原输入表导入的方法使壳程序能够正常运行。

首先回顾一下关于输入表的部分内容，它的结构如下：

[image]

自建输入表就是要将在外壳程序中用到的API调用，以这种结构构造出来并将所构造的输入表的RVA，作为加壳后的PE输入表RVA，写入PE头。下面以KERNEL32为例，说明实现方法如下：

[image]

可以看出，除了IMAGE_IMPORT_DESCRIPTOR结构外，其他部分也是按照输入表结构中的各项构造的，如IMAGE_IMPORT_BY_NAME结构、输入表名称表（INT）等。

由于自建的输入表很多项目都是相对于ImportTable的偏移的。所以，加壳程序在加壳时，要重定位各项目是以ImportTable为基准的。

外壳输入表RVA =（ImportTableBegin-ShellStart0）+ 外壳程序入口的RVA。

对自建的输入表中以ImportTableBegin为基准的项目都加上（ImportTableBegin-ShellStart0），这样就完成了重定位。当PE装载器装入加壳后的PE文件时，就会根据自建的输入表，将Kernel32模块的相应函数地址填入上面的输入地址表处。这样在外壳程序中，通过使用CALL[外壳入口点VA＋(AddressFirstShellStart0)]的方式调用Kernel32.dll的GetProcAddress。

16.3.3　外壳引导段

本章加壳的主程序是用Visual C＋＋来编写的，但外壳部分（shell.asm）是用32位汇编来写的。因此在加壳时就存在一个主程序如何调用shell.asm的变量问题。

具体实现如下：

（1）在32位汇编中对供高级语言调用的变量或子程序，使用PUBLIC声明。

（2）在Visual C＋＋6.0中调用时，使用“extern "C" DWORD para”声明。

例程中的相关程序如下：

[image]

这样可以直接在Visual C＋＋里调用shell.asm变量了。

接着是编译问题，具体参考附录B中的“四、在Visual C＋＋工程中使用独立汇编”。

首先将shell.asm填加到Visual C＋＋工程的Source files中，在Source files中的shell.asm上单击右键-＞选择Setting-＞选中Custom Build页。

命令行：c:\masm32\bin\ml/c/coff /Fo$(IntDir)\$(InputName).obj$(InputPath)

输出：$(IntDir)\$(InputName).obj

如果要生成调试信息，可以在命令行中加入“/Zi”参数，还可以根据需要生成.lst和.sbr文件。如果没有把MASM安装在C盘，则要做相应的修改。

在外壳代码设计时还会遇到一个数据寻址的问题。先来简单看一下程序编译时是如何进行数据寻址的。假设源程序中有一个变量名为“sum”，有一句代码为“mov eax,sum”，则编译后为“mov eax,[????????]”，其中的“????????”为变量sum在内存中的地址，由于程序

每次载入的地址都是固定的，所以执行时不会有什么问题，但是外壳代码遇到的却不是这种情况。外壳的代码是事前设计好的，当对不同的程序加密后，执行时外壳代码所在的内存地址都是不同的，如果也像一般程序那样编写，那么在运行时就无法正确找到变量，运行就会出问题。那如何解决呢？可以先设法得到程序中某个位置的地址，然后根据那些变量相对于此位置的偏移找到变量。

举例如下：

[image]

在本例中通过一个CALL指令自动将此CALL下一句（@@1）的地址入栈，然后用POP指令取出地址放入EDX，要取变量sum的值时通过“sum-@@1”取得变量相对于“@@1”的偏移，加上EDX就得到了变量的真实地址。这种做法在外壳中使用非常普遍，读者需要熟练掌握。

[image]

图16.4　外壳结构图

外壳部分的结构如16.4所示，其中以ShellStart为分界，之前的部分在压缩后的程序中是以非压缩的方式存在的，之后的部分是以压缩的方式存在的。外壳执行时先执行ShellStart0开始的部分，这部分的主要功能是将ShellStart开始的真正的外壳代码在内存中解压缩，并初始化一些数据。初始化完成后转移到ShellStart继续执行。ShellStart开始的代码是外壳的真正部分，它的主要功能是还原原始程序，另一个重要功能则是阻止破解者的跟踪、脱壳。所以一般来说这段代码会比较长，里面还有各种反调试器、反Dump的代码。将它以压缩的方式存储一方面可以减小文件尺寸，另一方面也有利于程序的安全性。下面我们就两段代码分别进行分析。

外壳第一段代码如下：

[image]

[image]

[image]

[image]

16.3.4　外壳第二段

第二段的主要工作是还原、初始化原程序，由于开始未对程序做特殊的处理，所以还原的工作主要就是解压缩。初始化工作主要指的就是初始化输入表，一般来说EXE文件载入基地址与程序的ImageBase都是相同的，所以重定位一般可以省略，如果要设计一个对DLL加壳软件的话，重定位这一步则是必不可少的。本来在第二段中应该有很多的代码用来抵抗脱壳者的攻击，这是加密的重中之重，也是判断一个外壳性能

的重要依据。但是为了突出本节的重点，在本例中这些代码都没有采用，在第二段只包含了最基本的用来还原原程序的代码。如果要设计一个有一定加密强度的加壳软件，反调试、反Dump、反监视等的代码是必不可少的，读者可以根据需要查看本书中其他章节的内容，在外壳第二段中加入相应的代码。不要怕麻烦，这种代码是多多益善的，哪怕只是消耗脱壳者的耐心，浪费他的时间也是好的。如果一个脱壳者在调试过程中不断遇到各种ANTI代码，稍有不慎就会导致前功尽弃，经过了数小时仍然不能看到希望的话，他多数会放弃的，那加密的目的也就达到了。不要希望通过某种简单的方法彻底阻止脱壳者的前进，任何防止调试、脱壳的方法对于有经验的解密者来说都是可以绕过的，或许用无数的代码破坏脱壳者的信心，最终拖垮脱壳者才是最彻底，也是最容易达到目的的解决方法。但是另一方面，过多ANTI代码的加入也会引起程序执行效率的降低。所以要编制一个优秀的加壳软件的话，应该在加密强度和执行效率之间找到一个比较好的平衡点。幸运的是，电脑硬件的飞速发展已经使得软件的执行效率不像以前那么重要了。

第二段代码如下：

[image]

[image]

[image]

[image]

[image]

[image]

[image]

[image]

[image]

16.4　将外壳部分添加至原程序

对程序处理的最后一步就是将外壳部分代码添加到原程序，一般的做法是给原程序增加一个独立的区块，将外壳的所有代码放入这个区块中。根据16.4可知，外壳由好几部分组成，所以在写入文件之前得先将它们装配起来。另外，原程序的一些重要数据也得保存在外壳的适当位置，比如程序原始的入口、程序的TLS表等。还有一点，一般程序的区块有很多在载入时是规定只能读不能写的，但是外壳要解压代码、还原程序就必须对这部分内存有可写的权限，这一般可以通过修改程序区块表中区块的属性来做到。下面就来分析程序。

[image]

[image]

[image]

[image]

[image]

[image]

至此，一个简单的加壳软件所需的各种功能的实现方法已经基本分析完了。至于这些功能如何完整地结合起来，读者可以根据附带的源码自行学习，此处仅讲一些要注意的地方。对PE文件的各项处理必须要注意其先后顺序，如果不注意的话可能会影响程序的执行效果，甚至使处理后的程序无法运行。比如资源处理的三步，如果将其顺序改为二、一、三，读者可以发现加壳后的文件会比原来的大，这就是第二步先处理后，对第一步的结果产生了不应有的影响。另外，防止程序被脱壳的方法除了尽量防止程序被脱壳者跟踪、调试外，尽可能地加强原程序和外壳的联系，也是一个不错的入手点。比如上面讲到的修改输入表，在初始化时不把正确的函数入口写入输入表，而让其指向外壳中的某段程序，必须经过这段程序的“翻译”，才能找到原始的入口。这就是设置了一条外壳与原程序联系的途径。有了这条途径，每执行一个函数外壳代码就得到一次系统的控制权，如果在其中加入ANTI代码的话，就可以更有效地保护程序不被攻击。此外，如果所写的外壳只是针对某一个程序，还可以把原程序的一部分功能代码放到外壳中。具体做法是，可以在程序本身做一个类似于引用外部DLL的调用，外壳在解密过程中先将功能代码解密到临时内存中，在初始化输入表时根据特定的函数名找出这些调用，并将它们初始化为指向临时内存中的功能代码，这样只有加壳后正常运行时这些功能才能使用，一旦脱壳就会使程序功能不全。当然将外壳与原程序联系起来的方法远不止这几种，两者之间联系得越多，完整、完美的脱壳也就会越困难。此处笔者只是提出一个思路，读者可以根据PE文件的结构特征、欲加密文件的功能特点设计各种不同的方法，自行发挥。

第17章　虚拟机的设计③

虚拟机是近几年刚刚兴起的名词，这里谈到的虚拟机和VMWare之类的虚拟机是不同的东西，它是一种基于虚拟机的代码保护技术。准确地说，这里谈的虚拟机其实是一种解释执行系统，例如Visual Basic 6中的Pcode编译方式，并且现在的一些动态语言如Ruby、Python、Lua和.Net等从某种角度讲也是解释执行的。要理解本章内容，需要对汇编指令和操作系统有一定的了解才行。

17.1　原理

虚拟机保护技术就是将基于x86汇编系统的可执行代码转换为字节码指令系统的代码，以达到保护原有指令不被轻易逆向和篡改，这种指令执行系统和Intel的x86指令系统并不在同一个层次上。比如说80x86汇编指令是在CPU里执行的，而字节码指令系统是通过解释指令来执行的，并且这里谈到的字节码指令执行系统是建立在x86指令系统上的。

字节码（Bytecode）其实就是指令执行系统定义的一套指令和数据组成的一串数据流。Java的JVM、.Net或者其他动态语言的虚拟机都是靠解释字节码来执行的，但它们的字节码之间并不通用，因为每一个系统设计的字节码都是为自己使用的，并不兼容其他的系统。

如图17.1所示是一个虚拟机执行时的整图概述，VStartVM部分初始化虚拟机，VMDispatcher来调度这些Handler，如果将其看成一个CPU的话，Bytecode就是CPU中所执行的二进制代码，VMDispatcher就是CPU执行调度器，Handler就是CPU中所支持的每一条指令。

[image]

图17.1　虚拟机图示

17.1.1　反汇编引擎

既然要将80x86指令转换为字节码，那么在做其他事情之前，将80x86指令反汇编为可读的结构是必然的工作。

反汇编引擎网上有现成的资源供参考，本节实例采用的反汇编引擎是OllyDbg提供的源代码，作者是Oleh Yuschuk，非常感谢他所做的工作。Oleh Yuschuk所提供的源代码的函数和结构笔者已稍做添加和修改，所以请尽量查看本书光盘映像文件中的源代码。

17.1.2　指令分类

这里将需要描述的x86指令进行分类，按功能可以分为4类：普通指令、堆栈栈指令、流指令和不可模拟指令。

（1）普通指令包括算术指令、数据传输指令等；

（2）栈指令主要是push和pop等进行栈操作的指令；

（3）流指令是如jmp，jmpc，call，retn等会更改程序执行流程的

指令；

（4）不可模拟指令，顾名思义，就是无法再次模拟的指令了，如int3，sysenter，in，out等，这类指令只能用其他方式处理。

按操作数可以分为：无操作数指令、单操作数指令、双操作数指令、多操作数指令。表17-1是分类指令表（不一定完全）。

表**17-1**　指令分类

[image]

其中，多操作数其实可以写为双操作数的形式，所以也可以在实现时将其归类为双操作数中。

当前主流动态语言的虚拟机是基于两种模式的：Stack-Based模式和Regiser-Based模式。除了Lua5现在已经改为Register-Based模式，其他语言现在还是Stack-Based模式。Stack-Based模式的优点在于字节码指令长度短，用到的指令比Register-Based模式更少。关于它们的差别，更多的是在编译器上。本节所讲述的虚拟机稍有一些不同，其是将汇编指令编译为字节码，不过仍然可以借鉴Stack-Based模式的思想。

17.2　启动框架和调用约定

在讲解Handler之前，有必要先说一下启动框架，因为它们之间是互相调用的，并且是相辅相成的，所以它们之间需要有一种代码约定。请先看启动框架是如何设计的。

17.2.1　调度器VStartVM

VStartVM过程将真实环境压入后有一个VMDispatcher标签，当Handler执行完毕后会跳回到这里形成了一个循环，所以VStartVM过程也可以叫做dispatcher（调度器）。

VStartVM首先将所有寄存器的符号压入堆栈，然后esi指向字节码起始地址，ebp指向真实堆栈，edi指向VMContext，esp再减去40h（这个值是可以变化的）就是VM用的堆栈地址了。换句话说，这里将VM的环境结构和堆栈都放在了当前堆栈之下200h处的位置上了。因为堆栈是变化的，在执行完跟堆栈有关的指令时总应该检查一下真实堆栈是否已经接近自己存放的数据了，如果是，那么再将自己的结构往更底下移动。然后从“movzx eax,byte ptr [esi]”这句开始，读字节码，读出一个字节，然后在JUMP表中寻找相应的Handler，并跳转过去继续执行。具体代码如下：

[image]

调用方法：

[image]

在这里看到了几个约定：

- edi = VMContext
- esi = 当前字节码地址
- ebp = 真实堆栈

这是整个执行循环都要遵守的一个事实，一般情况下谁也不应该将这些寄存器另做他用。另外，edi指向的VMContext存放在栈中而没有存放在其他固定地址或者申请的堆空间中，是因为考虑到多线程程序的兼容。假如有一个需要虚拟化的函数可能会被多个线程调用，而这时存放在固定的地址上就会出错，因为只能保存一个线程的环境结构。当然使用分配堆空间的API来为每一个线程都创建一个存放环境结构的空间也未尝不可，但虚拟机只是将汇编指令转换成虚拟指令来执行而已，使用API的话就会使其依赖操作系统而失去了兼容性，所以这里选择了堆栈。

17.2.2　虚拟环境：VMContext

VMContext即虚拟环境结构，存放了一些需要用到的值：

[image]

为什么这个环境唯独缺少了esp寄存器呢？因为esp寄存器的值已经被放在真实的ebp寄存器中了，VStartVM将所有的寄存器都压入了堆栈。所以，首先应该使堆栈平衡才能开始执行真正的代码，为此，设计了一个Handler VBegin来做这项工作。

17.2.3　平衡堆栈：VBegin和VCheckEsp

平衡堆栈：VBegin和VCheckEsp的代码如下：

[image]

执行这个Handler之后，堆栈就平衡了，就可以开始继续执行真正的代码了。但是，因为将VMContext结构存放在当前使用的堆栈更靠下面的一部分，所以应该避免出现VMContext结构被覆盖的情况。

[image]

图17.2　VMContext环境结构在堆栈中的分布

如图17.2所示，当堆栈被压入数据时，总会在某条指令之后改写VMContext的内容，因为这个原因设计了VCheckESP Handler。代码如下：

[image]

一些可能会涉及堆栈的Handler在执行后跳转到VCheckESP判断esp是否接近VMContext所在的位置，如果是就将VMContext结构复制到更远的位置存放。

17.3　Handler的设计

这里说的Handler，并不是Windows中的句柄，而是一段小程序，或者说是一段过程，是由VM中的调度器来进行调用的。

Handler分两大类：一类是辅助Handler，另一类是普通Handler。辅助Handler是一些更重要的、更基本的指令；普通Handler的功能是用来执行普通的x86指令的。

17.3.1 辅助Handler

辅助Handler除了VBegin这些维护虚拟机不会导致崩溃的Handler之外，就是专门用来处理堆栈的Handler了。请看下面几个Handler：

[image]

[image]

有了上述专门处理堆栈的Handler之后，就可以像图17.3一样设计普通x86指令的Handler了。

[image]

图17.3 普通x86指令的Handler

17.3.2 普通Handler和指令拆解

图17.3表达的意思是指令由普通Handler来处理，而源操作数和目的操作数都由堆栈Handler来处理。这样做的好处是，不必为指令的每一种形式都写一个模拟的Handler。

例如，add指令的形式通常有“add reg,imm”、“add reg,reg”、“add reg,mem”、“add mem,reg”等写法。如果将操作数都先交给堆栈Handler处理，那么执行到vadd Handler时，已经是一个立即数存放在堆栈中了，vadd Handler不必去管它从哪里来，只需要用这个立即数做加法操作即可。

先来实现一个vadd的指令：

[image]

请看下面的指令是如何转换为伪代码的：

[image]

转换为：

[image]

再来看这句：

[image]

转换为：

[image]

下面这句转换稍微有点复杂，因为源操作数是一个内存数，而内存数的真实结构是“[imm＋reg*scale＋reg2]”这样的。

[image]

本节对Oleh Yuschuk反汇编引擎做了修改，使其可以得到这些信息，具体信息请参考源代码。

这行指令可以转换为：

[image]

这就是add指令的多种实现，读者可以发现无论是哪一种形式，都可以使用vadd来执行，只是使用了不同的堆栈Handler，这就是Stack-Based虚拟机的方便之处。

17.3.3　标志位问题

标志位是一个麻烦的问题，稍有不慎就可能导致程序崩溃，并且难以调试。在x86中，涉及标志运算的指令有很多，如adc，add，and，bsf，bsr，bt，btc，btr，bts，cld，cli，cmc，cmovcc，cmp，cmps，cmpxchg，cmpxchg8b，daa，das，dec，div，idiv，imul，inc，jcc，mul，neg，not，or，rcl，rcr，rol，ror，sahf，sal，sar，shl，shr，sbb，scas，setcc，shld，shrd，stc，std，sti，sub，test，xadd等。其中有的指令是设置标志，有的指令是判断标志，所以在相关Handler执行前恢复标志位，执行后保存标志位。举个简单的例子，stc指令是将标志的CF位置为1：

[image]

这样操作之后就能保证代码中的标志不会被虚拟机引擎所执行的代码所改变。

17.3.4　相同作用的指令

在x86指令集中，为了提升性能或者其他原因，可以看到一些不同的指令其实可以用同一种指令去实现。比如如下两条指令：

[image]

虽然它们使用了不同的指令，但是它们的目的是相同的，这样的指令还有sub和dec。另外一些位运算指令也可以相互变换，不过位运算的变换可能会涉及标志位，使标志位的结果不同，因此有的地方指令变换时需要谨慎，但不是大问题。如果将这些x86指令这样化简之后，那么便不用对每个指令都做一个Handler来描述了。

17.3.5　转移指令

转移指令有条件转移、无条件转移、call和retn。这里先讲解前两类转移指令。

实现时可以将esi指向当前字节码的地址，esi指针就好比真实CPU中的eip寄存器，可以通过改写esi寄存器的值来更改流程。无条件跳转jmp的Handler比较简单：

[image]

条件转移jcc指令稍微有一点麻烦，因为它要通过测试标志位来判断是否需要更改流程，不过这里其实可以采取一些技巧。

读者会发现转移指令jcc和条件传输指令cmovcc高度匹配，请看表17-2中的一些比较。

表**17-2**　同等功能指令

[image]

基本上所有条件跳转指令都有相应的CMOV指令来匹配，感谢Intel设计出了这样一个指令，这样就好办了。这些指令可以这样设计：

[image]

字面上稍有不同的一个条件跳转指令是jecxz，对应的指令是cmovz。可以这样设计：

[image]

17.3.6　转移跳转指令的另一种实现

上节的条件转移是利用了cmovcc和jcc指令的相似性来模拟跳转，这样做太过简单，并且太过暴露，成了虚拟机的一个鸡肋。

这里有另外一种方法来模拟跳转，请看第4章表4-4中关于Jcc指令的描述。条件转移是根据标志位来判断是否需要跳转，那么模拟转移指令时判断标志位就行了。如图17.4所示是80x86的标志寄存器。

[image]

图17.4　80x86的标志寄存器

图17.4描述了标志位在标志寄存器中的位置，现在需要测试的标志位和所在位置都已经知道，就可以模拟条件跳转了。下面以jae指令为例：

[image]

这样调用这条指令：

[image]

这个指令首先得到标志位，然后和1做and运算（取CF位），cmove指令是判断ZF标志是否为0，为0就改变esi指向的地址。jae只是判断CF位，这里再以jbe做一个例子：

[image]

其他的跳转指令的实现也只是检测其他不同的标志位，没有太多的不同。

17.3.7 call指令

call和retn指令虽然也是转移指令，但是因为它们的功能不一样，所以被分开讲解。首先，虚拟机设计为只在一个堆栈层次上运行。请看如下代码：

[image]

其中第1、2、4条指令都是在当前堆栈层次上执行的，而call anotherfunc是调用子函数，会将控制权移交给另外的代码，这些代码是不受虚拟机控制的。所以碰到call指令，必须退出虚拟机，让子函数在真实CPU中执行完毕后再交回给虚拟机执行下一句指令。看起来，vcall这个Handler设计起来稍微有点麻烦，其实不然，call指令先压入下一句汇编代码的地址，然后跳到目标函数。下面的代码和它等同：

[image]

如果想在退出虚拟机后让anotherfunc这个函数返回后再次拿回控制权，可以更改返回地址，来达到继续接管代码的操作。在一个地址写上这样的代码：

[image]

这是一个重新进入虚拟机的代码，theNextByteCode代表theNext之后的代码字节码。只需要将theNext的地址改为theNextVM的地址，即可完美地模拟call指令了。vcall的伪代码如下：

[image]

17.3.8 retn指令

retn指令和其他普通指令不一样，retn在这里被虚拟机认为是一个退出函数。retn有两种写法：一种是不带操作数的；另一种是带操作数的。比如：

[image]

第一种retn形式先得到当前esp中存放的返回地址，然后再释放返回地址的堆栈并跳转到返回地址；第二种比前一种多了一个步骤，即在释

放返回地址的堆栈时再释放操作数的空间。vRetn的Handler设计如下：

[image]

17.3.9　不可模拟指令

不可模拟的指令前面也有提及，在这里任何不能识别的指令都可将其划分为不可模拟指令，碰到这类指令时，只能与vcall使用一种方法，即先退出虚拟机，执行这个指令，然后再压入下一个字节码的地址，重新进入虚拟机。

17.4　托管代码的异常处理

如果只是通过模拟跳转指令就想控制程序的流程是不够的，因为还有一种会打乱流程的情况，那就是异常处理。因此必须挟持原有的异常处理，才能绝对地控制程序的流程执行。关于编译器级的SEH更详细的资料，请参考其他文献。异常处理是不太可能完美解决的，只能针对编译器来进行模拟。

17.4.1　VC++的异常处理

VC编译器已经将Win32异常处理封装。如图17.5所示是VC 7编译器生成的栈帧布局。Scopetable是一个记录（record）的数组，每个record描述了一个__try块，以及块之间的关系。

[image]

图17.5　SEH3 Stack Layout

Scopetable的结构如下：

[image]

MSVC 2005的编译器为SEH帧增加了一些缓冲区溢出保护。完整的栈帧布局如图17.6所示。

[image]

图17.6　SEH4 Stack Layout

SEH4 Scopetable基本上和SEH3一样，只是加了一个Cookie头。没有什么不同，只要找对Scopetable的偏移就行了。

[image]

请看下面这样一个代码例子：

[image]

下面是对应的反汇编代码：

[image]

[image]

[image]

[image]

func1的汇编代码的第一句到“mov large fs:0,esp”这一句组成了这样一个结构：

[image]

其中，执行“push -1”之后，[ebp-4] = -1，也就是TryLevel = -1，代表未进入try块，这里有一个问题，那就是Scopetable这个数组没有数量，即不知道到底有多少个__try块。

[image]

上面两句则代表进入try block 0后，又进入了try block 1，当出现异常后，ExceptionHandler处理程序被执行，然后ExceptionHandler处理程序通过trylevel找到指向的try块在pScopeTable数组中搜索异常处理程序，即pScopeTable[trylevel].FilterFunc或pScopeTable[trylevel].HandlerFunc。现在好办了，知道了pScopeTable数组之后，就可以得到每一个异常处理程序的真实地址了。但还有一个小问题，现在并不知道pScopeTable数组有多少项，或者说并不知道有多少个try block。有两种方法来得到数组大小。第一种：暴力搜索pScopeTable，一直找到后面有一项的FilterFunc和HandlerFunc都为错误的地址时，就可以确定数组大小了；第二种：使用_trylevel的某种特征，比如通常情况下为-1（SEH3），通常所在的位置在ebp-4处，也可以通过计算异常代码和堆栈位置相互的关系来确定_trylevel的堆栈位置，找出所有对其赋值的常数，最大的那个常数应该就是数组的大小了。第一种简单有效，第二种比较复杂，且都不一定可靠，也不一定只有这两种办法，所谓八仙过海，各显神通了。

找到了_func1_scopetable数组中所有异常处理函数的地址后，就可以为每一个异常处理函数生成一个托管过程的代码，比如为func1 FilterFunc生成一个托管代码如下：

[image]

然后将scopetable- > FilterFunc的地址替换为func1_FilterFunc_Stub的地址，当出现异常时就会被调用，这时就再次进入了虚拟机，执行FilterFunc的字节码了。HandlerFunc也是一样。

17.4.2 Delphi的异常处理

Delphi的异常处理也经过了封装，描述其内部构造的文献很少，本节所述内容并未找到权威文献加以证明，请读者用怀疑的眼光看本节。

请看下面一段Pascal代码：

[image]

相应的汇编代码：

[image]

[image]

可以发现，Delphi所封装的异常处理不像VC那样带有一个数据结构，而是一句跳转指令跳转到了@HandleAnyException。

请看@HandleAnyException的代码：

[image]

[image]

[image]

[image]

一般情况下会执行到这里：

[image]

由此可以发现，Delphi的SEH异常封装似乎与VC的异常封装相比要简单得多，但是为了虚拟机代码要绕过这个封装却比VC稍微烦琐一点。既然HandleAnyException执行后跳转到Win32异常处理程序地址加5的位置（即下一句指令），那么可以生成一个下面这样的托管代码。

[image]

这一节主要描述了如何模拟VC和Delphi的异常处理机制，其他高级语言对异常处理的挟持也大同小异。

17.5 小结

这个虚拟机是将汇编指令转换成字节码来模拟执行的，因为汇编指令和字节码之间的特性不同，使得并不能完美地模拟汇编指令，这也是为什么无法将直接用汇编写的比较具有技巧性的代码成功转换为字节码执行的原因。另外，还有一些指令，比如jmp eax，这种代码比较模糊，并不确定要跳转到哪个地址。碰到这种指令，的确没有什么好办法，但是好在高级语言编译器似乎也不会编译出这种代码。

本章主要对虚拟机的大概框架、Handler设计、指令拆解和异常处理挟持作了详细的描述，而对于如何使用高级语言去实现并未作过多讲解，因为关于代码的设计不属于本章的范畴。本章附带源代码是笔者为参加看雪论坛2007年的CrackMe&ReserveMe大赛而匆忙设计的一个虚

拟机，它还缺少很多东西，如异常处理机制等，只能算是一个业余的作品，读者可以研究这个源代码以设计出一个更灵活的、更强大的虚拟机。

注释

① 本章由林子深编写。
② 本章由印豪编写。
③ 本章由冯典参与编写。

第8篇　PEDIY篇

- 第18章　补丁技术
- 第19章　代码的二次开发

“补丁技术”介绍了文件补丁和内存补丁技术，同时重点讲解了SMC技术在补丁方面的应用。学习补丁是一件很有意思的事情。

“代码的二次开发”主要是讲述如何在没有源码和无接口的情况下扩充可执行文件的功能，这一技术非常的实用。

第18章　补丁技术

在逆向工程中，经常需要改变程序原有的执行流程，使其增加或者减少一些功能代码，这就需要对原文件进行补丁。补丁分文件补丁和内存补丁两种。文件补丁就是修改文件本身某个数据，达到一劳永逸的效果。顾名思义，内存补丁是在内存中打补丁、修改，确切地说，是对正在运行的程序的数据进行修改，以达到某种效果。

18.1　文件补丁

文件补丁直接修改可执行文件或其功能模块的二进制数据，使其满足需求。实现起来很容易，对修改前和修改后的目标程序代码进行比较，把不同之处记录下来，然后写一个程序实现补丁功能。其优点如下：

（1）理论简单，容易实施。只需通过基本的十六进制数的操作就可完成任务。

（2）有很多现成的补丁工具供选择，填入几个参数，就可以做出自己的补丁器。

以第5章的EnableMenu为例讲述文件补丁的制作，需要修改代码如下：

[image]

程序的流程如下：

• 使用CreateFileA函数打开目标程序。

• 检查打开的文件是不是目标文件。可以检查文件大小，或者随机检查一些字节。

• 使用SetFilePointer函数把文件指针移动到指定位置。

• 使用WriteFile函数将数据写进磁盘文件中。

补丁源代码主体部分如下：

[image]

[image]

18.2　内存补丁

文件补丁虽然实现简单，但也有其局限性，如果待补丁文件被加壳，压缩或有完整性校验，则补丁不能顺利应用。正因为有了如此的一些不便之处，所以，在文件补丁技术以外，还需要一种更加高阶、隐蔽的方法，也就是内存补丁。

内存补丁的总体思想就是在某个时刻（解压、校验或某种情况发生以后），在目标程序的地址空间中修改数据，因此也被称为Loader，每次使用时都需要调用程序运行。

下面将分别详细论述这几种内存补丁的制作方法，并提供尽可能多

的源代码供读者参考。

18.2.1　跨进程内存存取机制

就操作系统的设计原则而言，运行于同一个系统内的进程A和进程B之间应该是“绝缘”的，也就是说，一个进程的地址、指令、内存使用等信息对于另一个进程而言应该是完全透明的。说得更高级一点，操作系统应该对运行于其内部所有的进程进行“封装”。可是在软件的实践过程中，“封装”和“灵活”从来都是一对不可调和的矛盾体。在C＋＋语言中，为了达到这两者的平衡，引入了friend关键字，引入了public、private、protected等存取层级概念。同样的，对于操作系统而言，在对进程进行封装的同时，也需要提供一种在一定条件下可以实现的进程间互访的机制。作为最简单的“进程互访”动作，内存的存取是一个最基本也是最重要的功能。所以Windows提供了两个用于进程间互访内存的函数ReadProcessMemory、WriteProcessMemory，只要进程的足够权限打开以后，就可读写进程的地址空间，权限可以用VirtualProtect函数设置。

在整个运行过程中，Loader载入待补丁程序可以由CreateProcess函数完成，资源释放、清理工作可以由ExitProcess函数完成。整个流程的核心之处在于流程中间的Loader对创建出来的待补丁进程的控制和修改，其将要补丁的代码通过WriteProcessMemory函数写入目标进程适当的位置，其运行过程如图18.1所示。

[image]

图18.1　内存补丁执行流程图

以第2章的TraceMe演示内存补丁制作，用ASProtect为TraceMe加壳，加壳后的程序命名为“TraceMe_asp.exe”。通过调试得知，只要将4011F5h的判断改为NOP指令，该实例输入任何姓名及序列号皆可注册成功。

[image]

修改为：

[image]

由于TraceMe_asp加壳保护，不能直接修改。此处给出一段补丁程序的代码，首先创建一个进程，然后定时把目标进程挂起，读取目标进程的代码是到了补丁时机；否则继续让目标程序运行，运行一段时间再挂起判断，直到目标进程解码。读者也可以用FindWindow查找窗口名来判断是否解码。整个代码框架如下：

[image]

[image]

在上面的代码段中，ReadProcessMemory函数的作用是读取目标进程特定地址的内容，有了这些内容，就可以对目标进程做一些校验性的工作。因为补丁这种程序是与程序的代码精确对应的，所以，WriteProcessMemory之前，对目标进程进行校验是非常必要的！

读出来的内容通过校验后，就可以进入下一步的正式补丁工作了，在这把补丁数据放到WriteData指向的内存中，然后调用WriteProcessMemory函数将这些数据写入指定的地址。

这个方法的缺点很明显，在这个例子中，要补丁的地址是4011F5h，可是在这似乎没有比较完善的方法能够让目标进程在运行到这个地址之前完成补丁工作。只有一个“勉强而且丑陋”的方法就是CreateProcess后，马上Suspend目标进程，然后等补丁完成后，再进行Resume的操作。

以上的补丁方法显然不能满足要求。理想的状况是需要一种“通知机制”，也就是让程序当EIP = 4011F5h时，能够给调试进程发送一个消息，调试进程收到消息后，再进行补丁操作。更美好的状态是：被调试进程在发送过来的消息中，不仅包含了EIP的内容，还能包含其他一些感兴趣的内容。很显然，要实现这种“通知机制”光凭自己的努力是不够的，还需要操作系统提供一些更底层的支持。下面就看看Windows提供给的Debug API能够在这个上面发挥什么样的功用。

18.2.2　Debug API机制

本节需要一些Debug API基础，读者可以通过其他渠道获取这方面知识。要用Debug API实现上节所说的“通知机制”，需要利用Debug API的EXCEPTION_DEBUG_EVENT调试事件。该事件的运作机理是这样的：

① 进程A内部产生异常；

② 操作系统监测到A的异常情况，并将该异常的相关情况包装到EXCEPTION_RECORD结构中；

③ 查找正在对进程A进行调试的进程，并将第②步包装好的结构通过EXCEPTION_DEBUG_EVENT消息发送到查找到的进程。

④ 陷入循环的调试进程收到进程A的EXCEPTION_DEBUG_EVENT调试信息，调试进程解析调试信号所包含的信息，并进行相应的操作。

补丁时，需要在目标进程执行到该地址的时候，发送一个EXCEPTION_DEBUG_EVENT调试消息给调试进程，以使其能够对此做出相应的处理。那么如何才能够让目标进程给调试进程发送调试消息呢？方法有两个：

（1）CreateProcess后，将目标进程设置成Single Step运行方式。由于目标进程每执行一条指令，都要和调试进程进行一次通信，所以该方法可以取得对目标进程的完全控制权。单步控制程序的弊端也是显而易见的：大幅度降低目标进程的运行效率。

（2）修改目标进程的代码，用类似于“文件补丁”的方法，将需要发送调试信息的地址处的原指令改为INT 3，这样当目标进程执行到特定地址时，就会因为INT 3的缘故，给调试进程发送相应的调试信息。

两种方法相比较，优劣很明显，当然是方法2比较高效和灵活。但

是出于全面性的考虑，下面还是分别将这两种方法的实现介绍给读者。毕竟，在运行时，将目标进程的执行方式设置成Single Step给调试进程所带来的强大威力是其他所有方法都不具备的。很多高阶的Loader程序，如大部分的脱壳机都要用到这项技术，所以掌握这项技术也是非常必要的。

1．Single Step辅助机制

要使进程在Single Step模式下运行，需要利用Intel CPU的EFLAGS中的一个Single Step标志位。如图18.2所示是Intel CPU EFLAGS寄存器的示意图。

[image]

图18.2　Intel CPU EFLAGS寄存器的示意图

图18.2中的SF标志位就是Single Step标志了。只要将其设置为1，就能保证目标进程每执行一条指令就能与调试进程进行一次通信。将该位置位的代码可以如下所示：

[image]

由于VC编译器默认的结构对齐机制已经满足了4字节对齐的要求，所以在Regs的声明处笔者并未加上多余的align说明。而如果读者是用MASM这种非常低阶的程序设计语言来编写代码的话，那么就需要在变量声明的时候加上这样的地址对齐说明伪指令。比如：

[image]

目标进程处于Single Step状态时，每执行一条指令就会给调试进程发送EXCEPTION_DEBUG_EVENT调试信息，该信息的ExceptionRecord.ExceptionCode部分为EXCEPTION_SINGLE_STEP。在WaitForDebugEvent循环中，可以监测该消息，然后进行相应的操作。需要注意的是，收到这个消息后，如果还想继续让程序Single Step下去，是需要重新设置一次SF位的。

2．INT 3中断

INT 3是Intel系列CPU专门引入的一个用于表示中断的指令，目标进程只要一执行INT 3指令，就代表目标进程发生了异常，Windows会将ExceptionRecord.ExceptionCode部分设为EXCEPTION_BREAKPOINT，然后发送给调试进程。

有了如此方便的触发异常的方法，只需要将需要补丁的地址处的指令改成INT 3，剩下的一切事情，交给调试进程吧！而调试进程是可以用任何语言编制的，可以实现任意功能的程序。很明显，一旦控制权交到调试进程手中，就可以对目标进程“为所欲为”了。而且该方法可以应付多个INT 3的情况，只需要收到调试信息后，利用GetThreadContext函数得到目标进程的EIP，就可以知道当前正处于什么样的位置上了。

本节利用Debug API完成一个类似于keymake的内存注册机，目标软件是没加壳的TraceMe，该实例的序列号是明码比较，调用lstrcmpA函数比较序列号，此时EBP寄存器指向真序列号。代码如下：

[image]

Loader的执行方式可以分为三段：

（1）将目标地址40138Dh指令改写为CCh（也就是INT 3的机器码）。

（2）中断触发异常后，读取EBP所指向的数据，并调用MessageBox显示出来。

（3）目标进程退出时，将其被修改的指令复原，以保证原软件功能的不变性。

本节提供的Loader没有考虑加壳的情况，其实现主体代码如下：

[image]

[image]

[image]

利用INT 3与前面几种方法相比，作为调试进程能做的事情更多，使用起来也更方便。但是仍然需要直接修改目标进程的代码的能力，也就是第1步，目标地址指令改写这一步。但是在很多情况下，软件被外壳保护，或软件自己的CRC校验过分强大（最突出的如WinHex等），或者软件中有多层SMC或其他各种“代码动态生成”机制时，该方法都会不适用。看来对于前面提出的两大难题：

（1）在适当的地址中断，控制权交给调试进程；

（2）控制权交付后，允许调试进程获取目标进程中断时的环境属性。

第2个难题可以通过Debug API完美解决；而前者，需要更底层、更隐蔽的方法，也就是让CPU辅助来调试！

18.2.3　利用调试寄存器机制

Windows操作系统提供了两种层次的进程控制和修改机制：

- 跨进程内存存取机制；
- Debug API监控目标进程运行信息。

但是这两种层次的进程监控机制都是运行在“操作系统”层次之上的，对于一些校验特别强悍的程序来说，这两种机制均不能满足要求。幸运的是，自386以后，Intel公司已经在其CPU内部集成了Dr0～Dr7一共8个调试寄存器，并且对EFLAGS标志寄存器的功能也进行了扩展，使其也具有一部分调试的能力。所以最隐蔽和最通用的内存补丁制作方式应该是利用CPU内建的DrX调试寄存器的调试能力来对目标进程进行补丁。

从Intel CPU体系架构手册中，可以找到Dr X调试寄存器的介绍，参考第2章的图2.25。当要对401000h设置的时候，将Dr 0～Dr 3其中的一

个设置为401000h，然后在Dr 7中设定相应的控制位。这样当被调试进程运行到401000h时，CPU就会给调试器发送异常信息，调试器可以捕获该信息进行相应的操作。

下面来看看这个威力强大的“调试信息通知机制”如何应用。作为接收方，不能直接接收DrX调试寄存器发出来的中断/异常信息，Windows已经将这个调试信息包装到了Debug API体系中，每当DrX调试信息被触发时，ExceptionRecord.ExceptionCode部分都被设置成EXCEPTION_SINGLE_STEP，只需要在Debug API循环中接受这个消息就可以达到目的。

自Windows 2000起，CreateProcess后，没有办法在目标进程的入口点地址处中断，常见的解决办法有两种。

1．利用**Single Step**机制

使进程在Single Step模式下运行，每执行一条指令就会给调试进程发送EXCEPTION_SINGLE_STEP，当收到第一个EXCEPTION_SINGLE_STEP异常信号，表示中断在程序的第一条指令，即入口点。从而达到中断在入口点的目的。

本节用DrX调试目标程序TraceMe，中断到指定地址，并将序列号显示出来。由于是利用DrX寄存器中断，因此目标程序可以加壳。CreateProcess后的部分代码如下：

[image]

[image]

[image]

上面的代码的流程大致是这样子的：

① 调用CreateProcess函数创建目标进程TraceMe_asp.exe，在创建进程的时候，传递DEBUG_PROCESSIDEBUG_ONLY_THIS_PROCESS调试参数。

② 由于是以DEBUG_PROCESSIDEBUG_ONLY_THIS_PROCESS参数创建的目标进程，所以拥有对目标进程完全的调试权和控制权。调试程序的主体就是一个调用WaitForDebugEvent(&DBEvent, INFINITE)函数构成的循环，一直到目标进程退出后，调试进程才会退出该循环。

③ 将目录进程设置在Single Step模式下运行

④ 当程序运行第一条指令时，就会第一次发送EXCEPTION_SINGLE_STEP异常信号时，表示中断在程序入口点处。并且设置Dr0的值等于入口点地址，Dr7 = 101h，表示CPU执行到Dr0中的地址时，发出异常信号。

⑤ 第二次收到EXCEPTION_SINGLE_STEP异常信号时，表示已经到达了程序的入口，这时，设置Dr0的值等于地址40138Dh。

⑥ 第三次收到EXCEPTION_SINGLE_STEP异常信号时，表示已中断在40138Dh，此时清除断点，并将EBP所指向的内容读出。

整个程序的运行过程都没有对目标文件进行任何改动，却实现了接

近完美的“消息通知”机制，而这一切，都得利于CPU上面的DrX寄存器的强大功能！

当然，DrX寄存器作为寄存器级别的调试工具，其应用范围绝不止内存补丁这么一种，结合Windows操作系统的Debug API功能和一些底层的驱动设施，DrX可以在很多与底层紧密交互的场合发挥出巨大的作用。而这一切，由于与本书主旨相去甚远，就留待给读者朋友日后工作中自己探索。

2．利用ntdll.ntcontinue作为跳板

CreateProcess后，当程序执行到ntdll.ntcontinue时，再对程序入口地址进行设断操作。EliCZ（API Hook Server的作者）最先发现了这个跳板，然后经由yoda（PeEditor及LordPE的作者）的bpm example code才流传开。

完整的代码见光盘映像文件，代码的流程大致是这样子的：

① 调用CreateProcess函数创建目标进程TraceMe_asp.exe。

② 目标进程正式运行前，会给调试进程发送一个EXCEPTION_BREAKPOINT调试信息，在调试进程内，通过dwBPCnt计数器判断出接收到的是第一个调试信息。然后是用GetProcessAddress和GetModuleHandle得到ntdll.ntcontinue的地址，并且设置Dr0的值等于该地址，Dr7 = 101h，表示CPU执行到Dr0中的地址时，发出异常信号。

[image]

③当收到第一个EXCEPTION_SINGLE_STEP异常信号时，表示中断在ntdll.ntcontinue函数，在这里，要把Dr0设置成程序的入口地址，可是这时不能用SetContextThread，而要用ESP的地址作为跳板，来设置调试寄存器的内容。具体的算法，程序中已经很清楚。

[image]

④第二次收到EXCEPTION_SINGLE_STEP异常信号时，表示已经到达了程序的入口，这时，设置Dr0的值等于地址40138Dh。

⑤第三次收到EXCEPTION_SINGLE_STEP异常信号时，表示已中断在40138Dh，此时清除断点，并将EBP所指向的内容读出。

18.2.4　DLL劫持技术

当一个可执行文件运行时，Windows加载器将可执行模块映射到进程的地址空间中，加载器分析可执行模块的输入表，并设法找出任何需要的DLL，并将它们映射到进程的地址空间中。

由于输入表中只包含DLL名而没有它的路径名，因此加载程序必须在磁盘上搜索DLL文件。首先会尝试从当前程序所在的目录加载DLL，如果没找到，则在Windows系统目录中查找，最后是在环境变量中列出的各个目录下查找。利用这个特点，先伪造一个系统同名的DLL，提供同样的输出表，每个输出函数转向真正的系统DLL。程序调用系统DLL

时会先调用当前目录下伪造的DLL，完成相关功能后，再跳到系统DLL同名函数里执行，如图18.3所示。这个过程用个形象的词来描述就是系统DLL被劫持（hijack）了。

[image]

图18.3　DLL劫持技术演示

利用这种方法取得控制权后，可以对主程序进行补丁。此种方法只对除kernel32.dll、ntdll.dll等核心系统库以外的DLL有效，如网络应用程序的ws2_32.dll、游戏程序中的d3d8.dll，还有大部分应用程序都调用的lpk.dll，这些DLL都可被劫持。

利用第5章5.6.2节提供的CrackMeNet.exe来演示一下如何利用劫持技术制作补丁，目标文件用Themida v1.9.2.0加壳保护。

1．补丁地址

去除这个CrackMe网络验证方法参考第5章5.6.2节，将相关补丁代码存放到函数PatchProcess()里。例如将401496h改成：

[image]

补丁编程实现就是：

[image]

p401496这个数组的数据格式，可以用OllyDbg插件获得，或十六进制工具转换。例如Hex Workshop打开文件，执行菜单“Edit/Copy As/Source”即可得到相应的代码格式。

2．构建输出函数

查看实例CrackMeNet.exe输入表，会发现名称为“ws2_32.dll”的DLL，因此构造一个同名的DLL来完成补丁任务。伪造的ws2_32.dll有着真实ws2_32.dll一样的输出函数，完整源码见光盘映像文件。实现时，可以利用DLL模块中的函数转发器来实现这个目标，其会将对一个函数的调用转至另一个DLL中的另一个函数。可以这样使用一个pragma指令：

[image]

这个pragma告诉链接程序，被编译的DLL应该输出一个名叫SomeFunc的函数。但是SomeFunc函数的实现实际上位于另一个名叫SomeOtherFunc的函数中，该函数包含在称为DllWork.dll的模块中。

如果要达到劫持DLL的目的，生成的DLL输出函数必须与目标DLL输出函数名一样。本例可以这样构造pragma指令：

[image]

编译后的DLL，会有与ws2_32.dll同名的一个输出函数WSAStartup，实际操作时，必须为想要转发的每个函数创建一个单独的pragma代码行，读者可以用工具AheadLib或用其他办法，将ws2_32.dll输出函数转换成相应的pragma指令。

当应用程序调用伪装ws2_32.dll的输出函数时，必须将其转到系统ws2_32.dll中，这部分的代码自己实现。例如，WSAStartup输出函数构造如下：

[image]

其中，GetAddress()函数的代码如下：

[image]

[image]

[image]

编译后，用LordPE查看伪造的ws2_32.dll输出函数，和真实ws2_32.dll完全一样，如图18.4所示。

[image]

图18.4　伪造ws2_32.dll的输出表

查看伪造的ws2_32.dll中任意一个输出函数，例如WSACleanup：

[image]

会发现输出函数WSACleanup()首先调用GetAddress(WSACleanup)，获得真实ws2_32.dll中WSACleanup的地址，然后跳过去执行，也就是说，ws2_32.dll各输出函数被Hook了。

3．劫持输出函数

ws2_32.dll有许多输出函数，经分析，程序发包或接包时，WSAStartup输出函数调用的比较早，因此在这个输出函数放上补丁的代码。代码如下：

[image]

hijack()函数主要是判断是不是目标程序，如果是就调用PatchProcess()进行补丁。

[image]

伪造的ws2_32.dll制作好后，放到程序当前目录下，这样当原程序调用WSASTartup函数时就调用了伪造的ws2_32.dll的WSASTartup函数，此时hijack()函数负责核对目标程序校验，并将相关数据补丁好，处

理完毕后，转到系统目录下的ws2_32.dll执行。

这种补丁技术，对加壳保护的软件很有效，选择挂接的函数最好是在壳中没有被调用的，当挂接函数被执行时，相关的代码已被解压，可以直接补丁了。在有些情况下，必须用计数器统计挂接的函数的调用次数来接近OEP。此方法巧妙地绕过了壳的复杂检测，很适合加壳程序的补丁制作。

一些木马或病毒也会利用DLL劫持技术搞破坏，因此当在应用程序目录下发现系统一些DLL文件存在时，如lpk.dll，应引起注意。

18.3 SMC补丁

利用SMC能修改自身代码这个特点，可以对加壳程序直接进行补丁，效果相当于内存补丁。加壳程序执行时，都有一个将数据解压并将其写入原始映像基址处的过程，在代码刚恢复还没运行前，在外壳里插入一段补丁代码，对刚解压出来的数据补丁。

18.3.1 单层SMC补丁技术

本节以UPX外壳演示一下单层SMC补丁技术。用UPX对第5章的EnableMenu加壳，保存为MenuUPX.exe。恢复程序功能需要修改代码如下：

[image]

现在不脱壳，直接在原文件里增加代码去除补丁限制。思路就是外壳代码将程序在内存中解压完毕时，让其先跳到补丁代码处执行补丁，再回到原指令处继续正常工作，具体过程如图18.5所示。

[image]

图18.5 SMC补丁示意图

用OllyDbg打开MenuUPX，发现UPX外壳跳到入口点（OEP）处的代码形式如下：

[image]

这段外壳代码是以未压缩的形式存在于外壳代码里的，外壳执行到这里时，原始的代码数据已还原，因此可以用此处作为SMC起点。

在加壳的MenuUPX.exe里，找一空间存放补丁代码。要求文件解压后，这个空间不能被原文件覆盖。同时程序运行时，不能被调用，如全局变量的调用。空间可以在各个区块间隙中获得，也可对原文件增加一个区块。此例选择前一种方案。用LordPE查看其区块信息，如图18.6所示。

[image]

图18.6 查看区块信息

UPX加壳后已将区块重新组织，分别是UPX0、UPX1等。UPX外壳代码在UPX1里，被压缩的原始数据放在UPX1中。运行时，外壳将解压缩后的原始代码映射到UPX0。PE文件各区块之间总有些间隙，在UPX1区块与.rsrc区块之间有一段空白代码空间（填充的00），其文件偏移为40E0h～4200h。由于这段空隙是在外壳代码空间里，加壳程序运行到OEP时，外壳代码部分将不会被执行了，因此这段空隙很适合放补丁代码。

被修改的UPX1区块属性必须可读写。幸运的是，一般加壳软件的区块是可读可写的。SMC补丁代码如下：

[image]

最后，用OllyDbg将修改后的结果保存到文件。

18.3.2　多层SMC补丁技术

一些外壳多层嵌套加密压缩，即在第一层代码中解密还原第二层代码，而第二层代码则解密第三层代码，依此类推。这就必须用多层SMC代码补丁才能达到效果。

用ASPack对EnableMenu.exe加壳，命名为“Menu_ASPack.exe”。用LordPE查看其区块信息，如图18.7所示。.aspack区块是外壳部分，其末尾的空白部分可以存放补丁代码。OllyDbg加载，查看.aspack区块尾部，将BB40hh（RVA）作为补丁空间。

[image]

图18.7　查看区块信息

如要正确SMC，必须确定需修改的语句“4011E3 6A01 push 1”在内存何处被解压。用OllyDbg加载Menu_ASPack后，在数据窗口对4011E3h下硬件写断点。

同时查看数据窗口4011E3h，直到代码被还原。此时程序外壳程序的代码如下

[image]

程序会中断在repz movsd语句处，向下不远处是个判断是否解压完毕的语句，在此之后让它跳到空白代码处。即做如下修改：

[image]

40BB40h处的补丁代码如下：

[image]

因为40A1A2h处的代码是动态生成的，磁盘文件中此处数据是加密的，所以必须再次用SMC技术补丁此处。首先确定这行代码在内存何处被解压。在数据窗口对40A1A2h地址下内存写断点或硬件写断点，重新

用OllyDbg加载实例运行。会中断如下指令处：

[image]

用十六进制工具可以在加壳的MenuASPack文件里找到上述代码的机器码，意味着可以直接修改程序了。选择何处代码执行SMC操作很关键，经分析，在40A5CEh一行时，已跳出了循环，40A1A2h代码被还原。所以选择40A5CEh处跳到空白代码处。代码修改如下：

[image]

40BB59h这段SMC代码是补丁40A1A2h处的指令，使40A1A2h处指令如下：

[image]

在OllyDbg中键入补丁代码：

[image]

将上述修改保存到磁盘文件里，就可完成此次的补丁了。一般外壳的代码区段属性是可读写的，如果被修补的地址不可写，此时必须调用VirtualProtectEx函数将其属性设置为可写。本例是两层SMC，有些外壳可能要多层SMC才能成功，操作方法类似两层，一层层地修补，一直到外壳的可见代码为止。由于补丁时，改变了原程序的指令，一些外壳会对代码进行完整性检查，因此遇到这种情况，必须找到校验处，将原始的校验值返回给外壳。

18.4 补丁工具

如果不喜欢编程，也可用补丁制作工具制作同样效果的补丁文件。补丁制作工具的种类较多，比如经典的文件补丁CodeFusion等，这些工具使用比较简单，参考一下帮助文档就能掌握。本节以dUP这款优秀的补丁制作软件为例，简单讲述一下补丁工具的使用。

dUP支持偏移补丁、查找与替换补丁、注册表补丁等，并可将这些方式混合，支持文件补丁和内存补丁，功能灵活强大，是目前一款流行的补丁制作工具。

dUP支持自定义界面，单击Settings标签，设置dUP的运行环境和补丁界面，如设置图标，定制皮肤，定制窗口形状等。

1．制作偏移补丁

补丁具体制作步骤在“Patch Data”选项卡里，单击“New Project”按钮，设置补丁程序的一些说明信息。单击“Add”按钮，添加补丁方式，dUP的各类补丁就在这选择。选择“Offset Patch”方式，然后在主界面里选择刚添加的“Offset Patch”补丁方式，单击“Edit”按钮，对其编辑，如图18.8所示。

[image]

图18.8　偏移补丁设置

对未加壳的程序制作文件补丁，选择“Normal File”模式，将补丁的文件偏移地址通过Add按钮填进来，也可比较（Compare Files）修改前后的文件来获得这些地址。

如果是要对加壳程序制作补丁，选择“VirtualAddress Mode”模式，以虚拟地址方式进行补丁。

添加补丁地址结束后，返回主界面，单击“Creater Loader”按钮创建内存补丁或单击“Creater Patch”钮创建文件补丁。其中文件补丁支持加壳程序，其原理是在原程序添加一段代码，运行时创建一个线程，监视指定地址数据，然后适机补丁。读者可以用本节的MenuASPack做个试验。

2．查找与替换补丁

在补丁方式里，选择“Search & RePlace Patch”，其可以在被补丁的程序中搜索指定的机器码，并替换成所需要的值，如图18.9所示。

[image]

图18.9　查找与替换补丁

所查找的机器码可能在程序里有重复，可以选择修改第几次出现的机器码进行补丁，建议查找的机器码尽可能长些，以降低重复的几率。如果没加壳，可以单击“check occurrence”按钮，其可查找重复机器码重复的频次。如果目标加壳，请将选项“Target is a compressed PE file”勾上。

3．注册表和附件补丁

如果需要往注册表里写相关数据，可以用dUP直接完成，将注册表文件*.reg导入即可。发布补丁时，可能需要附带一些文件，如DLL等，可以通过dUP的“Attached File”功能来完成。

第19章　代码的二次开发

本章主要是讨论在没有源码和无接口的情况下扩充可执行文件的功能，目标是二进制的EXE或DLL文件，需要用汇编实现相关功能，或构造一个接口，调用其他语言实现功能。该技术主要是修改扩充PE结构功能，对PE文件进行DIY，所以大家也称其为PEDIY技术。

19.1　数据对齐

数据对齐是CPU结构的一部分，对齐的目的是为了提高CPU运行效率。当数据大小的数据模数的内存地址是0时，数据是对齐的。例如，WORD值应该总是从被2除尽的地址开始，而DWORD值应该总是从被4除尽的地址开始。处理未对齐数据时，x86 CPU本身可以进行调整，代价就是占用CPU资源。

在Windows中，凡是要与系统核心打交道的数据结构，在声明的时候，一定要让其地址4字节对齐；否则，任凭程序逻辑怎么正确，算法怎么精妙，一定得不到想要的结果。微软文档规定映像页面相关数据必须对齐排列，不足的地方补0。Windows上的PE文件里的数据都是按这个要求对齐的。例如，输入表里，RVA以DWORD（4字节）对齐。所以当手工或编程构造PE文件的输入表、输出表和重定位表等时，必须将数据对齐细节考虑进去。

19.2　增加空间

在某些情况下为了增加原文件的功能，需要一定的空间存放代码。如果代码量不大，可以放到区块间隙里；否则必须增加一个区块。

19.2.1　区块间隙

由于PE文件每个区块的大小必定等于磁盘对齐值的整数倍，而区块的实际代码或数据的大小不一定刚好是这么多，所以在不足的地方一般以00来填充，这就是区块间的间隙，具体参考第10章10.4.3节。

实例pediy.exe的FileAlignment值为1000h，其磁盘文件中区块的大小就是1000h的整数倍，由于每个区块数据实际大小（见VSize值）不足这个值，因此各区块末尾都有一段空白的空间可以用，如图19.1所示。可以用十六进制工具打开文件查看，会发现VSize值后的数据是00，这些就是区块间隙。

[image]

图19.1　查看区块信息

利用区块间隙时，必须注意区块属性（Characteristics），其指出了该区块是否可读写的问题。例如，查看本实例的.data区块属性，在相应区块名上单击右键，执行“edit section header”命令，再单击“Flags”按钮打开属性状态图，其状态为可写（Writeable），如图19.2

所示。

[image]

图19.2　查看区块属性

由于.data区块是可写的，其区块中的一些间隙可能会被程序中的变量所使用，如全局变量或变量的缓冲区等。因此使用可写属性区块的间隙时，必须注意这些问题。如果区块属性是只读的，一般来说相对安全。

19.2.2　手工构造区块

在补丁的代码量比较大的情况下，可以新增一个区块。本节将在pediy.exe文件尾部（5000h）处）增加一个大小为1000h的数据段。

手工构造区块能熟悉PE格式，实际操作时一般可用工具辅助。增加区块有三个工作要做：一是增加一个块头；二是增加块头指向的数据段；三是调整文件映像尺寸。

构造区块时，必须注意区块的对齐。如果区块不对齐，在Windows 9x系统上程序运行可能没问题，但在Windows 2000/XP下程序会出错，如图19.3所示。

[image]

图19.3　区块不对齐而出错

1．修正块表

块表（Section Table）位于PE文件头之后，块表由一系列的IMAGE_SECTION_HEADER结构排列而成，每个结构描述的是一个块。结构的排列顺序和它们描述的块在文件中的排列顺序是一致的。全部有效结构的最后以一个空的IMAGE_SECTION_HEADER结构作为结束。

现在增加一个块头，具体内容如下：

[image]

构造区块头时，一定要注意块对齐的问题。用十六进制工具在原块表后，将上面的数据填入，具体如图19.4所示。

[image]

图19.4　增加一个块头

增加了一个块头后，就要修正PE头6Ch偏移处NumberOfSections的值，将其由原来的4改成5，表示当前区块数为5。用LordPE查看修正过的块表会发现多了一个区块（见图19.5）。

[image]

图19.5　增加后的块头

2．增加数据段

有了区块头，但区块中无数据，程序还不能运行。用十六进制工具在文件尾部5000h处插入1000h大小的数据块，数据块内容为0。这样，pediy区块就指向了大小为1000h的数据块，可以在这增加所需要的代码了。

3．修正映像文件尺寸

因为扩大了原文件的尺寸，所以必须修正SizeOfImage的值，将其由原来的5000h改成6000h。

19.2.3　工具辅助构造区块

实际增加区块操作时，一般用工具辅助完成。使用LordPE打开文件，在区块的列表上，执行右键菜单中的“add section header”功能就可增加一个区块。如果LordPE选项里勾选了“autofix SizeOfImage”，则自动修正SizeOfImage的值。但数据段的内容，还需要十六进制工具完成。另一款PE工具CFF Explorer，这方面功能比较强大，能自动增加区块及所对应的数据内容。专门增加PE文件区块的工具还有ZeroAdd、Topo等，操作都很简单。

一般的软件PE头部分的区块表（Section Table）后是一段全0的空间，因此新增区块不会破坏文件。但也有一些软件的块表后的空间没有富余，就不能增加区块了。例如Windows自带的一些程序，在块表后面紧跟着就是BoundImport数据。用ZeroAdd处理具有BoundImport结构的程序时，会报出“PE头空间不足”的提示。解决办法是在数据目录表里将BoundImport的RVA和Size赋零，同时将BoundImport数据区清零，处理后就可正常增加区块了。

19.3　获得函数的调用

在扩充程序功能时，经常遇到调用的API函数不在输入表中。解决方法：一种方法是修改输入表结构，增加相应的API函数；另一种方法是用显式链接方式调用DLL相关函数。

19.3.1　增加输入函数

可以用十六进制工具修改输入表中IID的成员，增加新的输入函数。也可以用相关的PE工具增加输入函数，如LordPE等工具。

本例将对实例文件增加MessageBoxA函数调用。用LordPE工具打开光盘映像文件中的实例addapi.exe，单击“Directories”按钮打开目录表窗口，在“Import Table”域中单击[image]按钮打开输入表编辑窗口，然后在任意一个DLL文件上单击鼠标右键，执行“Add Import”命令打开增加函数窗口（见图19.6）。输入DLL文件名和函数名，单击“OK”按钮即可为原文件增加一个函数。LordPE将新增一个区块，来存放新增加的IID数据。

[image]

图19.6　LordPE增加输入函数

当汇编代码调用新增加的函数时，在LordPE的输入表编辑窗口选择相应的DLL文件，勾上“View always FirstThunk”选项（见图19.7），则ThunkRVA项的值就是该函数在IAT中的RVA地址。调用时代码为：CALL[基址＋ThunkRVA]。

[image]

图19.7　查看调用地址

MessageBox在USER32.DLL用户模块中，其ANSI版是MessageBoxA。根据参数调用的需要，在实例的.rdata区块选择一空白处，构造两个参数所需要的字符串，如图19.8所示。

[image]

图19.8　构造字符串

构造好MessageBoxA的相关函数，然后用call[404019]方式就可直接调用：

[image]

如果在OllyDbg里直接键入“call MessageBoxA”语句，程序也能运行，看起来没什么问题似的。代码如下：

[image]

此时直接调用了MessageBoxA函数的入口地址，称为硬编码。如果操作系统不同，或USER32.DLL版本不同，MessageBoxA的地址就不一定是当前值，程序就会出错。所以，正确的做法必须在输入表里加入MessageBoxA函数，由PE加载器来获得函数的地址，再调用。

19.3.2　显式链接调用DLL

在应用程序调用DLL中的函数之前，DLL文件映像必须被映射到调用进程的地址空间中。有两种方法可以达到这一要求：一是加载时的隐含链接；二是运行期的显式链接。

“增加输入函数”一节讲述的就是隐含链接。显式链接是通过LoadLibraryA(W)或LoadLibraryExA(W)将DLL文件映像映射到调用进程的地址空间中，函数返回的HINSTANCE值用于标识文件映像映射到的虚拟内存地址。如果DLL文件已被映射到调用进程的地址空间里，则可以调用GetModuleHandleA(W)函数获得DLL模块句柄。一旦DLL模块被加载，线程可以调用GetProcAddress函数获取输入函数的地址。LoadLibrary、GetProcAddress等本身也是API函数，如果原输入表没

有，就得在输入表里增加这些API函数；或参考一些病毒技术，在kernel32.dll里暴力搜索出这两个函数地址。

光盘映像文件中提供的实例addapi.exe，已有LoadLibraryA、GetProcAddress函数，现利用这两个函数调用MessageBox函数显示一个对话框。根据这几个函数所需要的参数，在数据区构造出所需要的字符串，如图19.9所示。

[image]

图19.9　构造字符串

首先调用LoadLibraryA函数获得USER32.dll基址，其结果返回到eax寄存器，再将结果放进堆栈，然后调用GetProcAddress函数获得MessageBoxA的地址，最后直接调用。

[image]

本方法需要输入表中存在LoadLibraryA、GetProcAddress等函数；否则，必须在输入表中增加这些函数。

19.4　代码的重定位

在PE文件里，涉及直接寻址的指令都是需要重定位的。Windows系统会尽量保证EXE程序加载到所需要的基址，因此重定位基本可以不考虑。不过对于DLL的动态链接库文件来说，Windows系统没有办法保证每一次DLL运行时提供相同的基地址。这样修补DLL文件时，也必须提供“重定位”的代码，否则原程序中的代码可能无法正常运行。

19.4.1　修复重定位表

重定位信息是编译器生成的并保留在PE文件的重定位表里，如果程序里新增需要重定位的代码，则需要在重定位表里增加新的项。

在实例CodeRloc.dll的输出函数_DisplayTextA@0里增加一段代码，当被调用时显示MessageBoxA窗口。用LordPE在CodeRloc.dll输入表里增加MessageBoxA的调用，其ThunkRVA为C019h。用Hiew将如下代码数据修改到CodeRloc.dll文件里。代码如下：

[image]

在扩充功能新增补丁代码时，必须保证一些重要的寄存器的值不能被破坏，如ESI、EBP等，简单的做法是用pushad、popad指令保存现场所有寄存器。

在没修复重定位表之前，运行加载DLL，在笔者的系统中其被加载到基址370000h上，此时新增的代码情况如图19.10所示，由于代码没有重定位，因此继续运行程序将会崩溃。

[image]

图19.10　代码没重定位的情况

这段补丁代码有三处需要重定位，其中地址为401013h处就是其中一句需要重定位的语句，当基址是默认的400000h时，这句代码是正确的。当被加载的是其他基址时，401014h所指向的数据需要重定位。归纳起来，需要重定位的三处地址（RVA）是1014h、1019h、1021h。

重定位表以1000h大小为一个段，在IMAGE_BASE_RELOCATION结构中，VirtualAddress的值就是1000h的整数倍。本例需要重定位的1014h、1019h、1021h，可以放到VirtualAddress为1000h的组里，整理后的TypeOffset数据如表19-1所示。

表**19-1**　计算重定位数据

[image]

有两种方法将新增的TypeOffset放进重定位表结构中。第一种方法是在重定位表中，在RVA为1000h的索引中插入TypeOffset，这种方法比较麻烦，插入新的数据后，其他数据要往后移动。第二种方法是新构造一个重定位表段（见图19.11），追加到原重定位表后面。

[image]

图19.11　新增加的重定位数组

用十六进制工具，在原重定位后面追加新的重定位表数据上去，构造时，注意各数据按照内存存放规律放置——低地址放低字节，高地址放高字节，如图19.12所示。

[image]

图19.12　追加新的重定位表数据

最后，用LordPE将Directories中Relocation项的Size调整为522h。单击[image]按钮，以可视化方式显示新增的重定表数据，在图19.13中Index为9的就是新增的结构。

[image]

图19.13　增加的重定位表数据

运行加载修复后的CodeRloc.dll文件，此时DLL被映射到内存370000h地址上，相关代码这次被重定位了，如图19.14所示。

[image]

图19.14　已重定位后的代码

19.4.2　代码的自定位技术

不通过重定位表，也可实现代码的重定位，其原理是利用CALL指令执行时会将返回地址压入堆栈，然后用POP指令将这个返回地址取出，就可实现代码的自定位了。观察下面这段代码：

[image]

这段代码可以获得自身的地址，运行时CALL指令将返回地址401005h压入堆栈，下一句POP指令将返回地址取出放入EDX，再接下去减去CALL指令长度5，得到当前代码的地址。

在16.3.4节已讲解了这一技术在外壳上的利用，可以很巧妙地解决直接寻址的重定位问题。观察下面这段代码：

[image]

用POP指令取出地址放入EDX，通过SUB指令取得代码重定位的偏移，加上EDX就得到了变量的真实地址。

用LordPE在CodeRloc.dll输入表里增加MessageBoxA的调用，其ThunkRVA为C019h。将如下代码数据输入到CodeRloc.dll文件里。代码如下：

[image]

这段代码可以不需要重定位表，自身完成重定位功能。401013h这句的“call 40101E”执行后，紧跟其下方的字符串“PEDIY”地址被压入堆栈了，相当于语句“push 401018”。MessageBoxA用语句“call[edx＋40C019]”调用，EDX中保存的是重定位的偏移。

19.5　增加输出函数

一般情况下，若需要实现某个函数功能，可以重新写一个DLL文件，然后在EXE文件里调用该DLL的输出函数。但在特殊情况下，需要在现有DLL文件里增加输出函数。

在输出表中，各输出函数名之间必须按字母升序排列，否则Windows会报告“无法定位到相关DLL文件上”的错误。在新增输出函数时，必须注意这点。例如，有两个输出函数，第一个函数的第一个字母应该比第二个函数的第一个字母小；如果第一个字母一样，第一个函数的第二个字母应该比第二个函数的第二个字母小。

本例为MenuLib.dll增加一个zounter()输出函数，假设将增加的函数执行代码放在.data的5B00h处。

增加输出函数首先要了解输出表的结构，具体参考PE一章。原输出表RVA为4920h，大小为5Ah。输出表的IMAGE_EXPORT_DIRECTORY结构内容见表19-2。

表19-2　IMAGE_EXPORT_DIRECTORY结构

[image]

现在准备追加一个输出函数上去，若在原IED结构上修改，很不方便。这里采用先增加输出函数名，然后再重新构造一个IED结构。用十六进制工具打开MenuLib.dll，在原输出函数末尾处追加zounter字符串。将4990h处作为输出表的新地址，根据IED结构定义，依次构造各数据区，具体如图19.15所示。

[image]

图19.15　重新构造输出表

其中，指向函数地址数组指针AddressOfFunctions为49C0h，这里是3个输出函数的调用地址，在这里再补上新增的zounter()输出函数调用地址5B00h。指向函数名字的地址表的指针AddressOfNames为49D0h，并将zounter字符串的地址497Ah填上。指向输出序列号数组的指针AddressOfNameOrdinals放在49E0h中，并增加一项序号“02”。新构建输出表的IMAGE_EXPORT_DIRECTORY结构内容见表19-3。

表19-3　IMAGE_EXPORT_DIRECTORY结构

[image]

用LordPE打开MenuLib.dll，修正数据目录表里输出表的地址为4990h。最后要做的就是在5B00h处增加zounter函数的代码。若数据需要重定位，则要修复重定位表，并将.data属性设置为E0000040h，表示该块可读、可写、可执行并包含已初始化的数据。

19.6　消息循环

Windows应用程序的运行以消息为核心，每个窗口具有一个窗口函数，窗口函数从Windows接收消息，并检查每一条消息，根据这些消息完成特定的操作。也就是说，这里是程序的控制中心。本节主要是与大家探讨Win32 SDK写的程序，其他如MFC等消息处理比较复杂，不做深入探讨。

19.6.1　WndProc函数

窗口函数被程序员们称为WndProc，是一个消息处理回调函数，对消息进行判断和处理，一旦有消息产生，就会调用该函数。MFC采用消息映射实现对消息的响应，将各种消息拐弯抹角地送到各个对应的WndProc函数，使得消息的处理更加隐蔽。

Windows调用WndProc时，传递的参数有4个：hWnd参数为窗口句柄，message参数定义消息的类型，wParam和lParam参数包含消息的附加信息。

光盘映像文件中提供的实例pediy.exe文件的窗口函数（WndProc）源码如下：

[image]

WndProc的主体是由一系列case语句组成的消息处理程序段，程序员只需根据窗口可能收到的消息在case语句中编写相应的处理代码即可。如果想为原程序增加功能，如增加菜单、按钮等功能，就必须在消息循环WndProc里插入相应的处理代码。

19.6.2　寻找消息循环

根据Windows程序处理方式，用合适的API下断，就能方便地定位到消息循环（WndProc）的处理代码。

1．利用**RegisterClassA(W)**或**RegisterClassEx A(W)**函数

程序创建窗口之前，必须首先调用RegisterClass注册一个窗口类。该函数的参数是一个指向类型为WNDCLASS的结构指针，WNDCLASS结构的第二个成员lpfnWndProc指向WndProc。

RegisterClass函数原型如下：

[image]

WNDCLASS结构如下：

[image]

OllyDbg加载实例pediy.exe，用RegisterClassA设断，中断如下：

[image]

在数据窗口查看RegisterClassA的参数指向的WNDCLASS结构，如图19.16所示。WNDCLASS结构中lpfnWndProc指向的值为40112Eh，这就是WndProc地址。

[image]

图19.16　查看WNDCLASS结构

如果用IDA反汇编分析，其能自动分析出WNDCLASS结构初始化过程。代码如下：

[image]

2．利用**SendMessageA**函数

在某些程序中单击其菜单或按钮时，Windows会调用SendMessageA(W)函数发送一个WM_COMMAND消息给应用程序，该消息的wParam参数就是按钮或菜单的ID号。中断后，跟踪程序的处理过程就可发现WndProc处。

3．利用菜单或按钮对话框

当单击菜单或按钮时，会弹出一些对话框，如果能设断拦截，也可找到WndProc。

例如，单击pediy.exe关于窗口，会调用MessageBoxA函数打开版权窗口，用其设断也能来到WndProc消息循环里。

4．利用系统消息循环

当Windows处理完WndProc后，会重新返回到系统中，此时跟进，会来到系统消息循环处理代码处。不同系统，这个地址不同，笔者的Windows XP SP2的代码如下：

[image]

直接在系统消息循环处理中设置断点，如本例在77D18721h地址处下断，可以快速定位到WndProc，很适合MFC多个消息循环的程序。如果程序比较大，消息循环比较多，可能会频繁中断，此时可以用OllyDbg的log记录功能，将所有的WndProc记录下来分析。

5．利用GetWindowLongA(W)函数

编程获取本进程内窗口的窗口过程用GetWindowLong实现很简单，直接调用：

[image]

19.6.3　WndProc汇编形式

若在WndProc里设断，会发现Windows不停地调用该段代码，以处理程序的各类消息。Windows调用WndProc传递的参数有4个：hWnd、message、wParam和lParam。WndProc部分代码如下：

[image]

本例中，在WndProc代码里（40112Fh以后）可以用下面的变量调用其参数：

[image]

[image]注意：由于各类程序传递参数的方式不一样，因此具体问题具体分析。例如，有些程序用ESP来传递参数。

程序编译方式不同，CASE语句实现的代码略有不同。例如，按文件体积最小优化编译时一段样例代码如下：

[image]

19.7　修改WndProc扩充功能

本节将为pediy.exe程序增加三个功能：一是使菜单“File/Exit”命令生效；二是具有打开文本文件功能；三是具有保存文本文件功能。

19.7.1　扩充WndProc

如果想增加程序的菜单、按钮等功能，就在WndProc里加入新的消息判断和事件代码。修改时不得破坏原有的消息循环和堆栈平衡。

用Visual C＋＋、eXeScope或资源黑客等资源编辑工具为pediy.exe增加Open、Save等菜单，各菜单的ID自定义值见表19-4。

表19-4　菜单ID值

[image]

增加的菜单不得有重复ID，否则这些菜单功能会一样，因为Windows是靠ID来判断用户单击的是哪个菜单。菜单项目建好后，但由于还没事件代码，因此执行菜单不会有反应。单击菜单后，Windows会发送一个WM_COMMAND消息给应用程序，WM_COMMAND消息的wParam参数就是菜单的ID值，应用程序判断消息的地方就是在窗口函数（WndProc）里。

WndProc判断菜单ID的代码如下：

[image]

这段代码就是判断“关于”菜单的代码。现在必须增加一段代码判断Exit、Open和Save菜单的ID。程序在401250h后的空间没有被代码占用，因此将增加的代码放在此处。键入如下代码：

[image]

19.7.2　扩充Exit菜单功能

程序可调用PostQuitMessage(0)函数退出，即WM_DESTROY消息的事件代码。程序中原有的退出代码如下：

[image]

方案确定后，可以修改程序，让其直接跳到PostQuitMessage(0)函数处。修改的代码如下：

[image]

19.7.3　扩充Open菜单功能

打开文件需要调用VirtualAlloc函数申请内存空间。Edit控件编辑的最大文本是64KB，因此本例中分配的缓冲区大小定为64KB。

[image]

如果不需要内存，调用VirtualFree函数即可释放内存。

[image]

分配内存后就调用打开文件的对话框，这就需要定义OPENFILENAME结构。现在需要一点空间存放OPENFILENAME结构。再次调用VirtualAlloc函数分配内存，大小为（sizeof(OPENFILENAME)＝76 bytes）。

因为申请的内存块的数据都是0，所以只需初始化OPENFILENAME结构里的相关字段，其他字段用默认的0填充。OPENFILENAME结构中需要初始化的项目如下：

[image]

首先选定一个空间来存放OPENFILENAME结构的文件筛选字符串："*.txt.*.txt.*.*.*.*"。一般建议在各块的尾部找空隙，各块的起始部分有时好像是空的，但存在全局变量使用的可能性。本例选择在.rdata块的2D00h处存放筛选字符串。为了保证程序能正常运行，用LordPE编辑.rdata块，使VSize = RSize = 1000h。

初始化OPENFILENAME结构后，调用GetOpenFileName函数获得文件名。再调用CreateFileA函数打开文件，并用ReadFile将数据读出，接着调用SetWindowTextA函数将内存中的数据显示到文本框中。文本框的句柄可通过Create Window创建Edit控件的返回值获得。创建Edit编辑框的代码如下：

[image]

本例用隐含链接方式调用API函数，因此用LordPE修改输入表以增加所需的输入函数，各API函数的具体调用地址如表19-5所示。

表19-5　新增的输入函数

[image]

在40128Dh处键入如下代码：

[image]

[image]

[image]

[image]

修改完毕后，建议在不同系统或硬件上测试一下程序的功能是否正常。因为在修改代码过程中，很可能会在某些地方考虑不周全，在不同平台上测试会将隐藏的问题暴露出来。

增加Save菜单的原理类似，关键是要懂得基本的Win32编程，在此就不重复讲述了。

19.8　增加接口

上一节给pediy.exe增加菜单功能的例子固然巧妙，但工作量太大，并且很容易出错。为什么不写个DLL来实现打开、保存的功能呢？如果将消息循环的代码扩展到DLL文件来处理，这样功能扩展性更好。如果有兴趣，还可实现打印功能。这样，只需要用少量汇编代码提供一个接口，功能的实现只需调用DLL，非常的方便。

19.8.1　用DLL增加功能

写一个DLL，增加MenuOpen和MenuSave两个输出函数，在pediy.exe里直接调用相关输出函数完成功能。

1．创建DLL文件

读者可以用自己熟悉的语言创建DLL文件，如ASM、C、Delphi等。本例用Visual C++创建DLL，创建的DLL输出两个函数：MenuOpen和MenuSave（程序及源码见光盘映像文件）。

当Visual C++以stdacll方式将C函数输出时，Microsoft的编译器会改变函数的名字，设置一个前导下画线，再加上一个@符号做前缀，后随一个数字，表示作为参数传递给函数的字节数。例如，输出函数：

[image]

经编译后，会变成_MenuOpen@4。当然，此时可直接利用_MenuOpen@4函数进行输出操作，但也可让编译器输出没有经过改名的函数。方法是为编程项目建立一个.def文件，并在该.def文件中加上类似于下面的EXPORTS节：

[image]

当链接程序分析.def文件时，发现_MenuOpen@4和MenuOpen均被输出。由于这两个函数名是互相匹配的，因此链接程序使用MenuOpen的.def文件名输出该函数。

2．调用DLL函数

可以用隐含链接或显式链接方法调用输出函数。调用时需注意函数的调用约定，如StdCall、C调用等。本例是StdCall调用，函数内平衡堆栈。

用LordPE打开pediy.exe文件，增加MenuOpen和MenuSave两个输入函数，具体见表19-6。

表19-6　输入函数

[image]

这两个函数的参数是pediy.exe的句柄。由上一节已知，句柄是通过[EBP+8]来传递的。因此键入如下代码：

[image]

19.8.2　扩展消息循环

消息循环WndProc是程序的控制中心，上节是在汇编状态下扩展WndProc对消息的判断处理，更好的办法是将消息循环延伸到DLL里来处理，这样灵活性更高。

思路是：在调用原WndProc前，先转到DLL处理相关的事件，处理完再转到原WndProc执行。现在对pediy.exe程序WndProc开始处设断，代码如下：

[image]

刚进入WndProc时的堆栈，如图19.17所示。

[image]

图19.17　WndProc入口堆栈情况

此时堆栈中有一个返回地址和WndProc的4个参数。由于是在此处扩展消息循环（MyWndProc），可以将堆栈中这5个值作为MyWndProc的参数。设计的MyWndProc原型如下：

[image]

扩展的MyWndProc和WndProc相比，多了一个参数reversed，这是为了直接用图19.17所示堆栈中的各参数。修改程序时，只需要在汇编状态下直接调用MyWndProc()即可，不需要另传参数，比较简单。

MyWndProc接管消息后，就可用高级语言来扩充各功能了。代码如下：

[image]

将上述代码编写到一个DLL文件peplug.dll里，输出MyWndProc函数。让pediy.exe在调用自己的WndProc消息前，先调用MyWndProc，完成后继续原消息处理。

用LordPE在peplug.dll输入表里增加MyWndProc的调用，其ThunkRVA为5017h。修改pediy.exe程序如下：

[image]

跳到空白处，执行MyWndProc()，然后再跳回原消息循环处理。

[image]

读者也可直接修改WndProc起始地址，将其指向MyWndProc()，再跳回WndProc，这样补丁代码比较简洁。

[image]

MyWndProc()修改如下：

[image]

为程序打造接口，用DLL来扩展程序功能，灵活性大，维护十分方便。进行接口设计时，一定要注意堆栈平衡和不破坏原有寄存器。

附录A　浮点指令

一般的汇编书里对浮点指令介绍得很少，因此本节简单地介绍一下浮点数。

1．浮点数据格式

在计算机中表达实数时要采用浮点数格式，Intel 80x87遵循IEEE浮点格式标准。IEEE浮点数的格式由符号位S、指数部分E和尾数部分M三部分组成（见图A.1）。

[image]

图A.1　浮点数据格式

• 符号（Sign）：表示数据的正负，在最高有效位，负数的符号是1，正数的符号是0。

• 指数（Exponent）：或称阶码，表示数据以2为底的幂。指数采用偏移码表示，恒为整数。例如，单精度格式8位指数的偏移基数为127，除去全0、全1两个编码外，其余的1～254编码表示阶码数值-126～＋127。

• 有效数字（Significand）：表示数据的有效位数，反映数据的精度。

[image] 单精度浮点数据格式是32位，符号位占1位，E占8位，M占23位。

[image] 扩展单精度格式，符号位占1位，E大于或等于11位，M占31位。

[image] 双精度浮点数据格式是64位，符号位占1位，E占11位，M占52位。

[image] 扩展精度浮点数据格式是80位，符号位占1位，E大于或等于15位，M大于63位。

（1）把浮点格式数据转换成实数表达式

设单精度浮点数的符号位是s，指数是e，有效数字是x，则该浮点数的值用十进制表示为：

$= (-1)^{s} \times (1+x) \times 2^{(e-127)}$

例如：49E48E68 h = 0100 1001 1110 0100 1000 1110 0110 1000 b

将它分成符号、指数和有效数字3部分后的结果为：

49E48E68 h = 0　100 1001 1　110 0100 1000 1110 0110 1000 b

其中：

• 符号位是0，即s = 0。

• 指数部分是100 1001 1，读成十进制就是147，即e = 147。

• 有效数字部分是110 0100 1000 1110 0110 1000，也就是二进制的纯小数0.110 0100 1000 1110 0110 1000，其十进制形式为

0.7855958938598632812 5，即x = 0.7855958938598632812 5。

这样，该浮点数的十进制表示如下所示：

$= (-1)^{s} \times (1 + x) \times 2^{(e-127)}$

$= (-1)^{0} \times (1 + 0.7855958938598632812 5) \times 2^{(147-127)}$

$= 1872333$

（2）把实数转换成浮点格式

例如：100.25 = 0110 0100.01 b = 1.10010001 b $\times 2^{6}$

其中：

- 符号位 = 0；
- 指数部分 = 127 + 6 = 133 = 10000101 b；
- 有效数字部分 = 10010001000000000000000 b；

这样，100.25表示成单精度浮点数为：

0 10000101 10010001000000000000000 b = 42C88000 h

2．浮点寄存器

组成浮点执行环境的寄存器主要是8个通用数据寄存器和几个专用寄存器，它们是状态寄存器、控制寄存器和标记寄存器。

（1）浮点数据寄存器

浮点处理单元有8个浮点数据寄存器（FPU），编号为ST0～ST7，图A.2所示的是OllyDbg寄存器面板显示的浮点寄存器。每个浮点寄存器都是80位，并以扩展精度格式存储数据。8个浮点寄存器组成首尾相接的堆栈，不采用随机存取，而是按照“后进先出”的堆栈原则工作，并且首尾循环，所以浮点寄存器常常称为浮点数据栈。

[image]

图A.2　OllyDbg寄存器面板显示的浮点寄存器

（2）浮点状态寄存器

浮点状态寄存器表明FPU当前的各种操作状态，每条浮点指令都对它进行修改以反映执行结果，其作用与整数处理单元的标志寄存器EFLAGS相当，如图A.3所示。

[image]

图A.3　浮点状态寄存器

C3～C0是4位条件码标志，其中C1表示数据寄存器栈出现上溢或下溢；C3/C2/C0是保存浮点比较指令的结果。

3．浮点操作

这里介绍几个常用的浮点指令。

（1）取数指令

取数指令从存储器或浮点数据寄存器中取得数据，压入寄存器栈顶st(0)。“压栈”操作致使栈顶指针减1，数据进入新的栈顶st(0)，原来的st(0)成为现在的st(1)，原来的st(1)为现在的st(2)，依此类推。数据进入

寄存器栈前由浮点处理单元自动转换成扩展精度浮点数。

• FLD m32r/m64r/m80r/st(i)：取存储器或st(i)中的浮点数，压入栈顶st(0)；

• FILD m16i/m32i/m64i：取存储器的整数，压入栈顶st(0)。

（2）存数指令

该组指令除执行存数功能外，还要弹出（pop）栈顶。“出栈”操作是将栈顶st(0)清空，并使栈顶指针加1，原来的st(1)成为现在的st(0)，原来的st(2)为现在的st(1)，依此类推。出栈指令的助记符都是用P结尾。

• FSTP m32r/m64r/m80r/st(i)：st(0)按浮点格式存入存储器或st(i)，然后出栈；

• FISTP m16i/m32i/m64i：st(0)按整数格式存入存储器，然后出栈。

（3）比较指令

浮点比较指令比较栈顶数据与栈顶的源操作数，比较结果通过浮点状态寄存器反映，见表A-1。

表**A-1**　比较指令的结果

[image]

• FCOM：浮点数比较，st(0)和st(1)比较；

• FCOMP：浮点数比较，st(0)和st(1)比较，并出栈；

• FICOM m16i/m32i：整数比较，st(0)与m16i/m32i比较。

在比较结果中，“不可排序”是指两个浮点格式数不能按照相对值进行比较排序的一种关系。由于浮点指令没有转移指令，所以需将比较结果C3/C2/C0转换到整数状态寄存器中，然后利用整数转移指令进行跳转。具体过程如下：

① 首先用“FSTSW AX”指令将浮点状态字存入整数寄存器AX；

② 再通过“SAHF”指令将条件码传送到整数状态寄存器的低4位。SAHF指令将包含C3/C2/C0的高字节送入整数状态字的低字节，正好对应ZF/PF/CF；

③ 最后利用jxx指令进行条件判断转移（用无符号条件转移指令）。

汇编形式如下：

[image]

4．浮点指令汇总表

首先对下面的指令做一些说明：

• st(i)代表浮点寄存器，所说的出栈、入栈操作都是对st(i)的影响；

• src，dst，dest，op等表示指令操作数，src表示源操作数，dst/dest表示目的操作数；

• mem8，mem16，mem32，mem64，mem80等表示内存操作数，后面的数值表示该操作数的内存位数（8位为一字节）；

• x←y表示将y的值放入x，例如，st(0)←st(0)-st(1)表示将st(0)-st(1)的值放入浮点寄存器st(0)。

（1）数据传递和对常量的操作指令

[image]

（2）比较指令

[image]

（3）运算指令

[image]

[image]

[image]

[image]

附录B　在Visual C＋＋中使用内联汇编

使用内联汇编可以在C/C＋＋代码中嵌入汇编语言指令，而且不需要额外的汇编和连接步骤。在Visual C＋＋中，内联汇编是内置的编译器，因此不需要配置诸如MASM一类的独立汇编工具。这里以Visual Studio .NET 2003为背景，介绍在Visual C＋＋中使用内联汇编的相关知识（如果是早期的版本，可能会有些出入）。

内联汇编代码可以使用C/C＋＋变量和函数，因此它能非常容易地整合到C/C＋＋代码中。它能做一些对于单独使用C/C＋＋来说非常笨重或不可能完成的任务。

内联汇编的用途包括：

- 使用汇编语言编写特定的函数；
- 编写对速度要求非常较高的代码；
- 在设备驱动程序中直接访问硬件；
- 编写naked函数的初始化和结束代码。

1．关键字

使用内联汇编要用到__asm关键字，它可以出现在任何允许C/C＋＋语句出现的地方。先来看一些例子。

- 简单的__asm块：

[image]

- 在每条汇编指令之前加__asm关键字：

[image]

- 因为__asm关键字是语句分隔符，所以可以把多条汇编指令放在同一行：

[image]

显然，第一种方法与C/C＋＋的风格很一致，并且把汇编代码和C/C＋＋代码清楚地分开，还避免了重复输入__asm关键字，因此推荐使用第一种方法。

不像在C/C＋＋中的“{}”，__asm块的“{}”不会影响C/C＋＋变量的作用范围。同时，__asm块可以嵌套，而且嵌套也不会影响变量的作用范围。

为了与低版本的Visual C＋＋兼容，_asm和__asm具有相同的意义。另外，Visual C＋＋支持标准C＋＋的asm关键字，但是它不会生成任何指令，它的作用仅限于使编译器不会出现编译错误。要使用内联汇编，必须使用__asm而不是asm关键字。

2．汇编语言

（1）指令集

内联汇编支持Intel Pentium 4和AMD Athlon的所有指令。更多其他处理器的指令可以通过_EMIT伪指令来创建（_EMIT伪指令说明见下文）。

（2）MASM表达式

在内联汇编代码中，可以使用所有的MASM表达式（MASM表达式是指用来计算一个数值或一个地址的操作符和操作数的组合）。

（3）数据指示符和操作符

虽然__asm块中允许使用C/C＋＋的数据类型和对象，但它不能使用MASM指示符和操作符来定义数据对象。这里特别指出，__asm块中不允许MASM中的定义指示符（DB、DW、DD、DQ、DT和DF），也不允许使用DUP和THIS操作符。MASM中的结构和记录也不再有效， 内联汇编不接受STRUC、RECORD、WIDTH或者MASK。

（4）EVEN和ALIGN指示符

尽管内联汇编不支持大多数MASM指示符，但它支持EVEN和ALIGN。当需要的时候，这些指示符在汇编代码里面加入NOP指令（空操作），使标号对齐到特定边界。这样可以使某些处理器取指令时具有更高的效率。

（5）MASM宏指示符

内联汇编不是宏汇编，不能使用MASM宏指示符（MACRO、REPT、IRC、IRP和ENDM）和宏操作符（[image]!、&、%和.TYPE）

（6）段

必须使用寄存器而不是名称来指明段（段名称“_TEXT”是无效的），并且，段跨越必须显式地说明，如ES:[EBX]。

（7）类型和变量大小

在内联汇编中，可以用LENGTH、SIZE和TYPE来获取C/C＋＋类型和变量的大小。

- LENGTH操作符用来取得C/C＋＋中数组的元素个数（如果不是一个数组，则结果为1）。
- SIZE操作符可以获取C/C＋＋变量的大小（一个变量的大小是LENGTH和TYPE的乘积）。
- TYPE操作符可以返回C/C＋＋类型和变量的大小（如果变量是一个数组，它得到的是数组中单个元素的大小）。

例如，程序中定义了一个8维的整数型变量：

[image]

下面是C和汇编表达式中得到的iArray及其元素的相关值。

[image]

（8）注释

内联汇编中可以使用汇编语言的注释，即“;”。例如：

[image]

因为C/C++宏将会展开到一个逻辑行中，为了避免在宏中使用汇编语言注释带来的混乱，内联汇编也允许使用C/C++风格的注释。

（9）_EMIT伪指令

_EMIT伪指令相当于MASM中的DB，但是_EMIT一次只能在当前代码段（.text段）中定义一个字节。例如：

[image]

（10）寄存器使用

一般来说，不能假定某个寄存器在__asm块开始的时候有已知的值。寄存器的值将不能保证会从__asm块保留到另外一个__asm块中。

如果一个函数声明为__fastcall调用方式，则其参数将通过寄存器而不是堆栈来传递。这将会使__asm块产生问题，因为函数无法被告知哪个参数在哪个寄存器中。如果函数接收了EAX中的参数并立即储存一个值到EAX中的话，原来的参数将丢失掉。另外，在所有声明为__fastcall的函数中，ECX寄存器是必须一直保留的。为了避免以上的冲突，包含__asm块的函数不要声明为__fastcall调用方式。

[image]提示：如果使用EAX、EBX、ECX、EDX、ESI和EDI寄存器，则不需要保存它。但如果用到了DS、SS、SP、BP和标志寄存器，那就应该用PUSH保存这些寄存器。如果程序中改变了用于STD和CLD的方向标志，必须将其恢复到原来的值。

3．使用C/C++元素

（1）可用的C/C++元素

C/C++与汇编语言可以混合使用，在内联汇编中可以使用C/C++变量以及很多其他的C/C++元素，包括：

• 符号，包括标号、变量和函数名；
• 常量，包括符号常量和枚举型成员；
• 宏定义和预处理指示符；
• 注释，包括“/**/”和“//”；
• 类型名，包括所有MASM中合法的类型；
• typedef名称，通常使用PTR和TYPE操作符，或者使用指定的结构或枚举成员。

在内联汇编中，可以使用C/C++或汇编语言的基数计数法。例如，0x100和100H是相等的。

（2）操作符使用

内联汇编中不能使用诸如“＜＜”一类的C/C++操作符。但是，C/C++和MASM共有的操作符（比如“*”和“[]”操作符），都被认为是汇编语言的操作符，是可以使用的。举个例子：

[image]

在内联汇编中，可以使用TYPE操作符使其与C/C++一致。比如，下面两条语句是一样的：

[image]

（3）C/C＋＋符号使用

在__asm块中可以引用所有在作用范围内的C/C＋＋符号，包括变量名称、函数名称和标号。但是不能访问C＋＋类的成员函数。

下面是在内联汇编中使用C/C＋＋符号的一些限制。

• 每条汇编语句只能包含一个C/C＋＋符号。在一条汇编指令中，多个符号只能出现在LENGTH、TYPE或SIZE表达式中。

• 在__asm块中引用函数必须先声明；否则，编译器将不能区别__asm块中的函数名和标号。

• 在__asm块中不能使用对于MASM来说是保留字的C/C＋＋符号（不区分大小写）。MASM保留字包含指令名称（如PUSH）和寄存器名称（如ESI）等。

• 在__asm块中不能识别结构和联合标签。

（4）访问C/C＋＋中的数据

内联汇编的一个非常大的方便之处是它可以使用名称来引用C/C＋＋变量。例如，如果C/C＋＋变量iVar在作用范围内：

[image]

如果C/C＋＋中的类、结构或者枚举成员具有唯一的名称，则在__asm块中可以只通过成员名称来访问（省略“.”操作符之前的变量名或typedef名称）。然而，如果成员不是唯一的，你必须在“.”操作符之前加上变量名或typedef名称。例如，下面的两个结构都具有SameName这个成员变量：

[image]

[image]

如果按下面方式声明变量：

[image]

那么，所有引用SameName成员的地方都必须使用变量名，因为SameName不是唯一的。另外，由于上面的pszWeasel变量具有唯一的名称，你可以仅仅使用它的成员名称来引用它。

[image]

[image]提示：省略变量名仅仅是为了书写代码方便，生成的汇编指令还是一样的。

（5）用内联汇编写函数

如果用内联汇编写函数的话，要传递参数和返回一个值都是非常容易的。看下面的例子，比较一下用独立汇编和内联汇编写的函数。

[image]

C/C＋＋函数一般用堆栈来传递参数，所以上面的函数中需要通过堆栈位置来访问它的参数（在MASM或其他一些汇编工具中，也允许通过名称来访问堆栈参数和局部堆栈变量）。

下面的程序是使用内联汇编写的。

[image]

使用内联汇编写的GetPowerC函数可以通过参数名称来引用它的参数。由于GetPowerC函数没有执行C的return语句，所以编译器会给出一个警告信息，可以通过#pragma warning禁止生成这个警告。

内联汇编的其中一个用途是编写naked函数的初始化和结束代码。对于一般的函数，编译器会自动帮我们生成函数的初始化（构建参数指针和分配局部变量等）和结束代码（平衡堆栈和返回一个值等）。使用内联汇编，可以自己编写干干净净的函数。当然，此时必须自己动手做一些有关函数初始化和扫尾的工作。例如：

[image]

（6）调用C/C＋＋函数

内联汇编中调用声明为__cdecl方式（默认）的C/C＋＋函数必须由调用者清除参数堆栈。下面是一个调用C/C＋＋函数的例子。

[image]

[image]提示：参数是按从右往左的顺序压入堆栈的。

如果调用__stdcall方式的函数，则不需要自己清除堆栈。因为这种函数的返回指令是RETn，会自动清除堆栈。大多数Windows API函数均为__stdcall调用方式（仅除wsprintf等几个之外）。下面是一个调用MessageBox函数的例子。

[image]

[image]提示：参数可以不受限制地访问C＋＋成员变量，但是不能访问C＋＋的成员函数。

（7）定义__asm块为C/C＋＋宏

使用C/C＋＋宏可以方便地把汇编代码插入到源代码中。但是这其中需要额外地注意，因为宏将会扩展到一个逻辑行中。为了不会出现问题，请按以下规则编写宏。

- 使用花括号把__asm块包围住；
- 把__asm关键字放在每条汇编指令之前；
- 使用经典C风格的注释（“/* comment*/”），不要使用汇编风格的注释（“; comment”）或单行的C/C＋＋注释（“// comment”）。

举个例子，下面定义了一个简单的宏。

[image]

乍一看，后面的3个__asm关键字好像是多余的。其实它们是需要的，因为宏将被扩展到一个单行中：

[image]

从扩展后的代码中可以看出，第3个和第4个__asm关键字是必需的（作为语句分隔符）。在__asm块中，只有__asm关键字和换行符会被认为是语句分隔符，又因为定义为宏的一个语句块会被认为是一个逻辑行，所以必须在每条指令之前使用__asm关键字。

括号也是需要的，如果省略了它，编译器将不知道汇编代码在哪里结束，__asm块后面的C/C＋＋语句看起来会被认为是汇编指令。

同样是由于宏展开的原因，汇编风格的注释（“; comment”）和单行的C/C＋＋注释（“// commen”）也可能会出现错误。为了避免这些错误，在定义__asm块为宏时请使用经典C风格的注释（“/* comment */”）

和C/C＋＋宏一样，__asm块写的宏也可以拥有参数。和C/C＋＋宏不一样的是，__asm宏不能返回一个值，因此，不能使用这种宏作为C/C＋＋表达式。

不要不加选择地调用这种类型的宏。比如，在声明为__fastcall的函数中调用汇编语言宏可能会导致不可预料的结果（请参看前文的说明）。

（8）跳转

可以在C/C＋＋里面使用goto转跳到__asm块中的标号处，也可以在__asm块中转跳到__asm块里面或外面的标号处。__asm块内的标号是不区分大小写的（指令、指示符等也是不区分大小写的）。例如：

[image]

不要使用函数名称当作标号，否则将转跳到函数中执行，而不是标号处。例如，由于exit是C/C＋＋的函数，下面的转跳将不会到exit标号处：

[image]

美元符号“$”用于指定当前指令位置，常用于条件跳转中。例如：

[image]

4．在Visual C＋＋工程中使用独立汇编

内联汇编代码不易于移植，如果程序打算在不同类型的机器（比如x86和Alpha）上运行，可能需要在不同的模块中使用特定的机器代码。这时候可以使用MASM（Microsoft Macro Assembler），因为MASM支持更多方便的宏指令和数据指示符。

这里简单介绍一下在Visual Studio.NET 2003中调用MASM编译独立汇编文件的步骤。

在Visual C＋＋工程中，添加按MASM的要求编写的.asm文件。在

解决方案资源管理器中，右键单击这个文件，选择“属性”菜单项，在属性对话框中，点击“自定义生成步骤”，设置如下项目：

• 命令行：ML.exe/nologo/c/coff"-Fo$(IntDir)\$(InputName).obj" "$(InputPath)"

• 输出：$(IntDir)\$(InputName).obj

如果要生成调试信息，可以在命令行中加入“/Zi”参数，还可以根据需要生成.lst和.sbr文件。

如果要在汇编文件中调用Windows API，可以从网上下载MASM32包（包含了MASM汇编工具、非常完整的Windows API头文件/库文件、实用宏以及大量的Win32汇编例子等）。相应地，应该在命令行中加入“/I X:\MASM32\INCLUDE”参数指定Windows API汇编头文件（.inc）的路径。MASM32的主页是：http://www.masm32.com，里面可以下载最新版本的MASM32包。

术 语 表

Anti_Debug	反调试
Anti_Dump	反转存
Anti_Loader	反载入
Anti_Trace	反跟踪
API	Application Programming Interface，应用程序编程接口
ASCII	美国信息交换标准码，ASCII码
Buffer Overflows	缓冲区溢出
CLR	Common Language Runtime，通用语言运行时
CrackMe	一些公开给他人调试解密的小程序
CRC	Cyclic Redundancy Checksum，循环冗余校验码
DDK	Device Drivers Kit，设备驱动程序开发包
Decompiler	反编译，将程序还原成高级语言的原始结构
Disassembler	反汇编，将机器语言转化成汇编语言
DPL	描述符特权级别
Dump	抓取内存镜像数据保存到磁盘
EP	Entry Point，文件执行时的入口点
Exploit	漏洞利用程序
ExploitMe	一种有漏洞的小程序，用来练习Exploit技术
Export Table	输出表，导出表，引出表
File Offset	磁盘文件偏移地址
GDT	Global Descriptor Table，全局描述符表
GUI	图形视窗程序
Handle	句柄
Hash	哈希算法，单向散列函数算法
Heap	堆
IAT	Import Address Table，输入地址表，导入地址表
IDT	中断描述符表
IID	输入表的IMAGE_IMPORT_DESCRIPTOR结构
ImageBase	基址，基地址
Import Table	简称IT，输入表，导入表，引入表
Inline Patch	同SMC
INT	Import Name Table，输入名称表
KeygenMe	一些供给别人尝试解密的小程序，要求做出它的keygen（序号产生器）
Loader	内存补丁
Module	模块
MSIL	微软中间语言
Nag窗口	警告提示窗口

Native-Compile	自然编译，编译器将高级语言转换为汇编代码
Obfuscation	混淆
OD	OllyDbg的简称
OEP	Original Entry Point，原始入口点
Overflow	溢出
pack	壳（音ké），一种专用加密软件
patch	为文件打补丁
Pcode-Compile	伪编译，编译器将高级语言转换为某种编码后，解释执行
PE	Portable Executable，Windows系统可执行文件格式
PEB	Process Environment Block，进程环境块
PEDIY	修改扩充PE结构功能，对PE文件进行DIY
Reflect	反射
ReverseMe	为练习逆向技术而写的（或特别构造的）小程序
Reversing	逆向，或称逆向工程（Reverse Engineering）
Ring0	操作系统内核模式
Ring3	操作系统用户模式
Rootkit	一种被设计的能得到最基本（高）权限的程序
RPL	请求特权级别
RVA	Relative Virtual Address，相对虚拟地址
SDK	Software Development Kit，软件开发工具包
SDT	ServiceDescriptorTable，服务描述表
Section	区块，区段，节
SEH	Structured Exception Handling，结构化异常处理
ShellCode	通称缓冲区溢出攻击中植入进程的代码
SMC	Self-Modifying Code的缩写形式，在一段代码执行之前先对它进行修改
Stack	栈
stolen bytes	指外壳将程序部分代码变形，并搬到外壳段
String Encoding	字符串编码
StrongName	强名称
TEB	Thread Environment Block，线程环境块
TLS	Thread Local Storage，线程局部存储
Unmanaged Code	非托管代码
unpack	脱壳
Virtual Address	简称VA，内存虚拟地址
VM	Virtual Machine，虚拟机
领空	指在某一时刻，CPU的CS:EIP所指向的某段代码的所有者

参 考 文 献

[1] Charles Petzold. Programming Windows. Microsoft Press, 1998

[2] John Robbins. Debugging Applications. Microsoft Press, 1999

[3] Gary Nebbett.Windows NT 2000 Native API Reference, 2001

[4] Everett N. McKay,Mike Woodring. Debugging Windows Programs. Addison-Wesley

[5] Bruce Schneier. Applied Cryptography. John Wiley & Sons, 1996

[6] Jeffrey Richter. Programming Applications for Microsoft Windows. Microsoft Press, 2000

[7] Matt Pietrek. Windows 95 System Programming SECRETS. IDG Books, 1995

[8] Eldad Eilam.Secrets of Reverse Engineering. Wiley, 2005

[9] Matt Pietrek.An In-Depth Look into the Win32 Portable Executable File Format

[10] Randy Kath.The Portable Executable File Format from Top to Bottom

[11] David Solomom.Inside Windows NT,2nd Edition

[12] FIPS 186-2.Digital Signatrue Standard. NIST 2000

[13] FIPS 197.Announcing the Advanced Encryption Standard. NIST 2001

[14] A public Key CryptoSystem and a Signature Scheme Based on Discrete Logarithms.TAHER Elgamal, 1985

[15] Media Crypt. Internation Data Encryption Algorithm Technical Description

[16] RFC1321.The MD5 Message-Digest Algorithm. R.Rivest, 1992

[17] David J. Wheeler, Roger M. Needham. "TEA, A Tiny Encryption Algorithm",

[18] "Guide to Elliptic Curve Cryptography", Darrel Hankerson,Alfred Menezes, Scott Vanstone, Springer 2003

[19] Cryptography and Network Security Principles and Practices, Third Edition William Stallings, 2003

[20] Making, Breaking Codes An Introduction to Cryptology, Paul Garrett, 2003

[21] Cryptography Theory and Practice (Second Edition), Douglas R. Stinson, 2002

[22] Professional C#(3rd),Simon Robinson,Wiley Publishing,

2004

[23]　Expert .NET 2.0 IL Assembler,Serge Lidin,Apress, 2006

[24]　Essential .NET Vollume 1: The Common Language Runtime, Don Box, Addison Wesley, 2003

[25]　C# to IL,Vijay Mukhi, BPB Publications, 2003

[26]　Distributed Virtual Machines: Inside the Rotor CLI,Gray Nutt, Addison Wesley, 2005

[27]　ECMA-334, C# Language Specification; ECMA-335, Common Language Infrastructure (CLI)

[28]　Applied Microsoft. NET Framework Programming, Jeffrey Richter, Microsoft Press, 2002

[29]　Common Language Infrastructure Annotated Standard, Jim Miller, Addison Wesley, 2003

[30]　看雪学院．软件加密技术内幕．北京：电子工业出版社，2004

[31]　Kris Kaspersky. 谭明金译．黑客反汇编揭秘．北京：电子工业出版社，2004

[32]　罗云彬．Windows环境下32位汇编语言程序设计．北京：电子工业出版社

[33]　周明德．保护方式下的80386及其编程．北京：清华大学出版社

[34]　杨季文．80X86汇编语言程序设计教程．北京：清华大学出版社

[35]　尤晋元，史美林等．Windows操作系统原理．北京：机械工业出版社，2001

[36]　侯杰．Windows系统编程奥秘

[37]　邹丹．www.zoudan.com．关于Windows 95下的可执行文件的加密研究

[38]　Lenus.浅谈脱壳中的Dump技术．看雪软件安全论坛

[39]　Peansen. 记事本功能增加方案．看雪软件安全论坛

[40]　CCDebuger.OllyDBG入门系列教程．看雪软件安全论坛，2006

[41]　CCDebuger. 浅谈程序脱壳后的优化．看雪软件安全论坛，2006

[image]

中国计算机网络安全应急年会
暨中国互联网协会网络安全工作年会

每年一度的中国计算机网络安全应急年会（简称CNCERT年会），旨在为国内外政府相关部门，电信、金融等产业界，以及科研、教育、学术机构提供一个全面交流的机会，在广泛的领域展示最新的研究成果和具有挑战性的实践与项目，促进跨学科的研究开发和分享新思考，积极参与国际互联网领域的合作、交流，促进中国互联网的健康发展。在CNCERT年会中，您将可与来自政府、企业、学术等各界的领导和顶尖专家进行面对面的交流，一般包括以下领域的内容：

- 在安全事件防御、检测和响应方面的先进技术；
- 在计算机与网络安全工具方面的最新进展；
- 交流计算机安全事件响应领域的观点、经验和解决方案

2009中国计算机网络安全应急年会（第6届）——诚邀您的参与

主办单位

国家计算机网络应急技术处理协调中心
中国互联网协会网络与信息安全工作委员会

大会网址

http://conf.cert.org.cn

联系方式

电话：010-88254013，15011478020
E-mail：bn@phei.com.cn；bining2008@sina.com MSN: cert_cn@163.com

协办单位

中国通信学会通信安全技术委员会

承办单位

电子工业出版社

会议时间

2009年5月5日～8日

会议地点

中国长沙

[image]

中国软件安全峰会
China Software Security Summit

现代社会中，我们所需的一切几乎都与计算机系统息息相关，例如：电力、交通、通讯、金融、电子政务等，都要依赖于高可靠性的计算机系统。但是近年来，越来越多的个案说明它们并不如我们想象的那么安全可靠。因而，软件安全已经成为我们这个时代一个备受社会关注的大问题。

针对软件安全日趋严重的问题，中国软件安全峰会汇聚软件企业、信息安全企业、政府相关部门、科研、学术和教育界的关注软件安全的人士进行交流，在广泛的领域展示和研讨最新的研究成果和各种实践方法，共同探索解决问题之道，促进中国软件产业的健康发展。

2009中国软件安全峰会——与您共享软件安全前沿话题

主办单位

电子工业出版社

会议网站

http://www.sinoit.org.cn

会议时间

2009年6月（大连）
2009年9月（北京）
2009年11月（上海）

联系方式

电话：010-88254012，15011478020
E-mail：sinoit@phei.cn; panxin@phei.com.cn

欢迎订阅《软件安全》电子期刊

每半月一期的《软件安全》电子期刊（www.sinoit.org.cn）于2008年12月创刊，旨在为IT一线专业开发人员提供高可读性和高实用性的技术文章，帮助您了解更多业内资讯和热点活动。

热点文章

◎揭开加密解密未来的神秘面纱
——看雪软件安全网站创始人段钢谈加密解密技术的发展趋势
◎是“屏黑”，不是“黑屏”
——微软大中华区首席安全顾问江明灶谈Windows正版增值计划
◎换一种方式做加密——著名程序员刘涛涛谈扭曲加密变换技术【★文章附后】
◎等级保护概述——著名等级保护专家陆宝华谈我国信息安全等级保护概况
◎走近虚拟机——McAfee研究员孙冰谈虚拟机技术和虚拟机安全

热门资源

◎.NET程序加解密技术——.NET程序保护专家　单海波【★文章附后】
◎扭曲变换加密技术——著名程序员　刘涛涛
◎Fuzz技术与软件安全性测试——著名软件安全专家　王清
◎国内外软件漏洞处理概况——国家互联网应急中心（CNCERT/CC）运行部副主任　周勇林
◎Web 2.0时代的网络安全——奇虎公司董事长助理　石晓虹

订阅方式

◎登录www.sinoit.org.cn，在线提交订阅信息。
◎将您的姓名、电子邮箱等信息发送到sinoit@phei.cn，并请注明“订阅电子期刊”。

了解更多精彩内容，敬请访问**www.sinoit.org.cn**

[image]

加解密所关注的.NET带来的改变

软件安全专家　单海波

一、PE文件结构的改变

.NET作为新的平台，肯定引进一些新的技术。我们讨论的是加解密，因此我们有特殊的关注点。第一是在PE结构上带来的一些改变。大家很清楚PE结构是Windows平台上可执行程序的结构，普通的PE文件保存的是机器指令的编码，图1是典型的普通的PE文件示意图。

[image]

图1　普通的PE文件

.NET下的PE文件，在.NET下叫程序集，其保存的是另外两种数据，叫元数据和MSIL，当然同时还保存其他的数据，但主要是元数据和MSIL中间语言。

在.NET下，PE结构中改变最大的属于.text节，它包含.NET的基本结构和数据，比如说CLR头，一些输入表、元数据等，这是.NET对PE文件的最大的改变。

讲到这里引出.NET中一个非常重要的概念，元数据。什么是元数据？字典上解释为描述数据的数据，用在.NET上也是非常合适的。首先它本身就是一个数据，0101；第二它描述了.NET程序的全部信息，一个.NET程序想正常运行，必须是一个合法的.NET程序，必须包含合法完整的元数据。因此我们说.NET程序是自描述的，运行的时候，程序集本身包含了所有信息，不需要寻找别的依赖性的信息（比如在注册表中）。

但是程序集的这种自描述的性质，使得反编译时的信息更加丰富，结合MSIL的特征，使得反编译的结果近似源代码。如图2所示，左边是一个反编译的图，再看看点击的方法，所有的名称都被显示了出来。所以说元数据还是挺可怕的。

[image]

图2　程序集的这种自描述性质使得反编译时信息更加丰富，结合MSIL的特性，使得反编译结果近似原代码

下面简单介绍一下元数据的基本结构，元数据是以数据流的形式保存在文件中，各种有不同的形式，一个是堆，一个是表，流是统称，而堆和表是存在形式。

图3表示了.NET下的六个元数据堆，第二个流是Blob，所有二进制数据都保存在Blob中，第三个是GUID，每个程序集唯一，这个比较简单。第四个流，US代表用户字符串，红色标示的是比较常见的，和加解密关系比较重要的，因为用户字符串中保存了比较关键的信息，。下一个#-，最重要的流，所有的堆存在于这个流中。

[image]

图3　.NET下的6个元数据堆

图4的倒数第二行介绍了所有的元数据堆，也就是.NET的元数据表，从00到44，大家看看这些名称基本上反映了.NET面向框架结构，基本上结构从名称中就可以看出来了。

[image]

图4　.NET的元数据表

二、统一了编程语言

第二个主要的变化是统一了编程语言。像我们编程做应用开发的时候，可以使用C＋＋、C#或者其他语言，但是这些语言经过静态编译以后，统一到MSIL。

MSIL的特点，第一它是高级语言，面向对象的，比汇编高级很多。第二它是是基于堆栈的运行机制。

来看一段示例代码（图5）。

[image]

图5　代码

第一行定义局部变量，名称为VAL，下面为MSIL指令。我们一句一句的看。首先初始堆栈为空，然后第三句，出栈，又入栈，然后10和1两个参数入栈，最后运算，结果入栈。这是MSIL最显著的特点，但是也给.NET加解密带来一个非常常用的操作，叫往复操作（round-tripping），就是拿到一个.NET的PE文件，我们反编译成源文件，在源码上进行操作以后，通过框架提供的ilasm，再编译成可执行文件，只有通过元数据和MSIL可以实现这种操作。

三、程序运行方式的改变

加解密所关注.NET的变化第三点我觉得是程序运行方式的改变。Windows不再直接负责程序的运行，而是由.NET框架进行管理。框架中JIT引擎负责在运行时将IL代码即时编译为本地汇编代码执行。.NET程序被称为托管程序，相反地，我们说的传统的Windows程序，就是非托管程序（或称为本地程序）。

下面简单介绍.NET的基本概念。

首先是我们常听到的程序集。程序集是.NET可执行程序的基本单元，程序集中间包含一个或多个模块，模块和程序集的区别是模块不含清单信息，所以模块是不能直接运行的，而程序集是可以直接运行的。在MSIL中编程时，程序集没有扩展名。

第二个是类型。面向对象大家非常熟悉了，差别不是很大，方法也是一样。类型有很多成员，其中最重要的是方法，因为加解密关注方法中的关键代码，所以方法是非常重要的成员。

然后是标识，就是怎么定位.NET中每一个元数据，在MSIL是一个32位的值，类似AABBBBBB的形式。比如方法元数据在元数据表中排第七个，从零开始计数的话，就是06，Main方法为第一个方法，排第一，因此Main方法的元数据标识就0x06000001。再比如某字符串为0X70000001，就是这样一个简单的标识。

最后，有一个概念叫签名，签名就是存储在Blob中的一段二进制数据，它的作用是描述特定元数据的性质。.NET下共有六个表包含签名项。

程序集的运行方式，即时编译（JIT）是.NET运行可执行程序的基本方式，也就是在需要运行的时候，将对应的IL代码编译为本机指令，传入JIT中的是IL代码，导出的是本机代码。所以，部分加密软件通过挂钩JIT进行加解密操作。

换一种方式做加密

著名程序员　刘涛涛

一、逆向的可怕

根据经验，如果努力一点，一人一天可以逆1000行C＋＋代码。直到上个月，我还刚刚完成了一个六万行C＋＋代码的工程。也就是说，你辛辛苦苦几年时间开发出来的软件，如果不加保护，可能会以每人每天1000行的速度被逆向全部代码，这是多么可怕的一件事情，尤其是驱动。我们知道，驱动是最富技术含量的，驱动开发是最难的，能够做驱动开发的人，都是顶尖高手，都是薪水很高的人。而恰恰驱动却是软件保护中一个问题最严重的地方。我们可以看到，我们的系统里面的那些驱动程序，个个短小。就是说，我们可能花费了很大的精力，很长的时间开发出来一个驱动，包括了我们软件的所有核心技术，但是它可能只有几十KB。一个有经验的逆向者，可能几天时间逆向出全部源码。

所以说被破解不可怕，可怕的是被逆向。如果你的软件被逆向了，就会冒出多个与你的软件功能相同的软件，你所有的技术就都不存在了。

二、变形的思考

虚拟机变形，很强大，很复杂，这也是它的缺点。它的变换并不神秘，并不可怕。你也可以构思一个虚拟机，可以大胆地构思，假如说我有一个CPU是8位的，那么这时候当你遇到一条32位的一条指令的时候，要想用你8位的CPU实现这个代码，就不得不编一长串的程序，完成32位下的一条指令。

它的倍率很高，但倍率不可调。我的选择，用多种低倍率的变换，迭代使用，这样倍率可随意调整，强度一点也不差。最重要的，我可以全文件低倍加密，重点部分高倍加密。

实际上要让代码难懂，并不需要那么麻烦。不就是要把代码扭曲吗？不就是要把代码混淆吗？难道非用虚拟机不可吗？我的意思是我们并不一定非要用虚拟机，我认为虚拟机变换不过是精心设计的一套变换，不过是一套变换，一套规律，尽管这套规律非常复杂，但是你毕竟

是一套规律，我只要有精力去学习，我肯定还是能搞懂的。我认为与其我们用一套复杂的规律去对代码做加密变换，倒不如我们用多套简单的变换来做同样的事情。

用轻量的变换，然后再对少数参数做重点的变化，这是我的思想。

所以如果说有人给我一个软件，说它是加壳的，我肯定说这个没问题，有办法解决它。如果有人给我一个软件，用虚拟机加密的，我也会说这个没问题，因为多数函数是明码，只有极少数是加密的。我把90%以上的函数都搞懂了，剩下的几个函数尽管是用虚拟机加密的，我知道入口点在哪里，我知道入口参数是什么，甚至我也可以把它当成一个黑箱，看看它返回什么，这时候我已经搞懂了。

但是如果有人给我一个用我说的方法加密的软件，每一个函数都加了密，每一个简单函数都需要费很大精力才能搞得懂，我会说这个东西没办法做了，这是我的思路。

在实际的软件分析过程中，如果你看见一个高手在分析一个软件，过程是什么样子的呢？是一步一步去理解这个软件。先是找一些最简单的函数，把它搞懂，比如这个函数是打开一个文件，这个函数是什么，肯定是先从这些最简单的函数着手，然后在这个过程当中，这个分析者逐步地增强信心，他的思路逐渐明确，他越来越有信心，这个软件最后被搞定了。但是如果按照我的思路的话，他是不断地受到打击，他是举步维艰，就像我上面说的，如果他分析了一天，分析出一个函数，发现是两个数相加，他会气坏了，不干了。

具体要做加密，我的做法是基本上是以函数为单位进行加密变换，我首先会对这个函数进行堆栈分析，就是我要知道它那些技术变量是怎么实现的，有一些冷僻的汇编代码，汇编指令是无法识别的。下一步进行标志分析，也就是标志寄存器。如果我们每一步加密变换，都小心翼翼不敢去破坏这个标志的话，太受约束了，这个变化就不会丰富多彩了。我们用高级语言，用C＋＋写出来的代码，绝大多数是不依赖于标志寄存器的。

实际上，这里使用的加密变换，不过是一个由简入繁的过程，应该说，每个人都可以编出自己的一套变换，不过碰巧这是我写出来的变换罢了。这里的每一个变换都不可怕，可怕的是我这种变换是可以迭代的。我起名字很简单，叫变换一、变换二、变换三，来了一个软件，准备加密他，来个一二一吧，先做一变换，再做二变换，再做一变换，变换之后这个代码就非常可怕了。

可以想像，只要我们这套将引擎开发完成了，要想增加新的变形方法，简直是易如反掌。我们可以召集几个程序员开会，限定你们十分钟之内，给我想两个加密变换。这个没有问题，由繁入简是唯一的，但是由简入繁方法太多了，经过一番复杂的运算，最后逆过来。所以说这个变换可以充分发挥我们的想象力。比方说你可以让你的EAX总是对真正的EAX取个反。

经过以上几步变换，代码已经很难懂，但还是顺序的。再加一个变换，用JMP指令打乱。这时候已经分不出我这个函数从哪里开始，到哪里结束。你找不到call，好不容易找到一个，结果是假的。这时候分析者连一个函数都找不到了。

有人说，你为什么不一条指令一跳呢？我觉得一条指令一跳代码的膨胀量太大了，2K的代码非要变成2MB，我希望我们的变换在前面的一二一里面有技术含量，用跳转太没有技术含量了，我每三五条跳转一下也就够了。

最后一步叫做缠绕。以上用JMP打乱还只是本函数内部打乱，如果把多个函数缠到一起，就无法从内存区域划分一个个函数。

通过以上多种变换方案的组合、迭代，我们可以自由选择控制加密深度，一个100KB的软件，我们可以加密为200KB，2MB，甚至200MB，而且没有垃圾代码，这200MB代码都是有用的，如果要理解它，就都要分析，海量的代码使分析变为不可能。

[image]

Broadview®
www.broadview.com.cn

《加密与解密（第三版）》读者交流区

尊敬的读者：

感谢您选择我们出版的图书，您的支持与信任是我们持续上升的动力。为了使您能通过本书更透彻地了解相关领域，更深入的学习相关技术，我们将特别为您提供一系列后续的服务，包括：

1．提供本书的修订和升级内容、相关配套资料；

2．本书作者的见面会信息或网络视频的沟通活动；

3．相关领域的培训优惠等。

请您抽出宝贵的时间将您的个人信息和需求反馈给我们，以便我们及时与您取得联系。

您可以任意选择以下三种方式之一与我们联系，我们都将记录和保存您的信息，并给您提供不定期的信息反馈。

1．电子邮件

您可以发邮件至**jsj@phei.com.cn**或**editor@broadview.com.cn**。

2．信件

您可以写信至如下地址：北京万寿路**173**信箱博文视点，邮编：**100036**。

3．读者电话

您可以直接拨打我们的读者服务电话：**010-88254369**。

在您选择的联系方式中，您还可以告诉我们更多有关您个人的情况，及您对本书的意见、评论等，内容可以包括：

（1）您的姓名、职业、您关注的领域、您的电话、E-mail地址或通信地址；

（2）您了解新书信息的途径、影响您购买图书的因素；

（3）您对本书的意见、您读过的同领域的图书、您还希望增加的图书、您希望参加的培训等。

如果您在后期想退出读者俱乐部，停止接收后续资讯，只需编写邮件"退订＋需退订的邮箱地址"发送至邮箱：market@broadview.com.cn即可取消该项服务。

同时，我们非常欢迎您为本书撰写书评，将您的切身感受变成文字

与广大书友共享。我们将挑选特别优秀的作品转载在我们的网站（**www.broadview.com.cn**）上，或推荐至**CSDN.NET**等专业网站上发表，被发表的书评的作者将获得价值**50**元的博文视点图书奖励。

我们期待您的消息！
博文视点愿与所有爱书的人一起，共同学习，共同进步！

通信地址：北京万寿路173信箱　博文视点（100036）　　电话：010-51260888
E-mail：jsj@phei.com.cn, editor@broadview.com.cn

[image]
www.phei.com.cn
www.broadview.com.cn

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：（010）88254396；（010）88258888
传　　真：（010）88254397
E-mail：　dbqq@phei.com.cn
通信地址：北京市万寿路173信箱
　　　　　电子工业出版社总编办公室
邮　　编：100036

增值服务

尊敬的读者：

感谢您选择我们出版的图书。鉴于本书容量有限，由本书作者撰写的与本书相关的以下内容将以电子文档形式提供。

★SoftICE调试器

★w32dasm的使用

★SmartCheck调试工具

★WKTVBDebugger调试工具

★浮点指令补充

★InstallShield反编译

您任意选择以下三种方式与我们联系，将获得文档下载地址和密码以及本书的最新修订内容。

1．短信

您只需编写如下短信：

A06644或B06644＋您的电子邮箱地址＋您的建议　发送到10666666789

（本服务免费，短信资费按照相应电信运营商正常标准收取，无其他信息收费）

为保证我们对您的服务质量，如果您在发送短信24小时后，尚未收到我们的回复信息，请直接拨打电话（010）88254369。

2．电子邮件

发邮件给market@broadview.com.cn，内容：书名＋您的联系方式＋您的需求＋您对本书的评价。

3．信件

您可以写信至如下地址：北京万寿路173信箱博文视点，邮编：100036。

另外，我们将从与我们联系的读者中选出幸运读者若干名进行抽奖，并赠送如下礼品。

一等奖：将获得价值600元的由电子工业出版社与看雪学院共同组织的2008～2009年度软件安全年会门票一张或者价值100元的“安全技术大系”图书。

二等奖：将获得价值50元的“安全技术大系”图书。

三等奖：参加“安全技术大系”读者见面会或者获得相关活动的视频资料。

（本活动最终解释权归电子工业出版社所有。）

博文视点愿与所有爱书的人一起，共同学习，共同进步。